Order this document by
DSP56307EVMUM/D
Rev. 1.4, 7/98

DSP56307EVM User’s Manual

Motorola, Incorporated
Semiconductor Products Sector
6501 William Cannon Drive West
Austin TX 78735-8598

MOTOROLA

This manual is one of a set of three documents. You need the following manuals to have complete product information: the family manual, the user’s manual, and the technical data.

Table of Contents

Chapter 1
Quick Start Guide

Chapter 2
Example Test Program

Chapter 3
DSP56307EVM Technical Summary

Appendix A
DSP56307EVM Schematics

Appendix B
DSP56307EVM Parts List

Appendix C
Motorola Assembler Notes

List of Tables

List of Figures

List of Examples

Chapter 1
Quick Start Guide

This section summarizes the evaluation module contents and additional requirements and provides quick installation and test information. The remaining sections of this manual give details on the DSP56307EVM design and operation.

1.1 Equipment

The following subsections list the equipment required to use the DSP56307 Evaluation Module (DSP56307EVM), some of which is supplied with the module, and some of which must be supplied by the user.

1.1.1 What You Get with the DSP56307EVM

The following material comes with the DSP56307EVM:

- DSP56307 Evaluation Module board
- *DSP56307EVM Product Brief*
- *DSP56307 Technical Data* (preliminary)
- DSP56307 Chip Errata
- *DSP56300 Family Manual*
- *DSP56307 Product Specifications, Revision 1.03*
- *DSP56307EVM User's Manual* (this document)
- Crystal Semiconductor CS4218 16-bit Multimedia Audio Codec Data Sheet
- Domain Technologies Debug-56K Debugger manual for Motorola 16- and 24-bit DSPs
- The required software:
  - — GUI Debugger from Domain Technologies (1 diskette)
  - — Assembler/linker software from Motorola (1 CD)

1.1.2 What You Need to Supply

The user must provide the following:

- PC (486 class or higher) with:
  - Windows95 or NT
  - Minimum of 16 Mbytes of memory
  - 3½-inch high density diskette drive
  - CD–ROM drive
  - Hard drive with 20 Mbytes of free disk space
  - Mouse
  - RS-232 serial port supporting 9,600–115,200 bit-per-second transfer rates
- RS-232 interface cable (DB9 male to DB9 female)
- Power supply, 7–9 V AC or DC input into a 2.1-mm power connector
- Audio source (tape player, radio, CD player, etc.)
- Audio interface cable with 1/8-inch stereo plugs
- Headphones

1.2 Installation Procedure

Installation requires the following four basic steps:

1. Preparing the DSP56307EVM board
2. Connecting the board to the PC and power
3. Installing the software
4. Testing the installation

1.2.1 Preparing the DSP56307EVM

CAUTION

Because all electronic components are sensitive to the effects of electrostatic discharge (ESD) damage, correct procedures should be used when handling all components in this kit and inside the supporting personal computer. Use the following procedures to minimize the likelihood of damage due to ESD:

- **Always handle all static-sensitive components only in a protected area, preferably a lab with conductive (antistatic) flooring and bench surfaces.**
- **Always use grounded wrist straps when handling sensitive components.**
- **Do not remove components from antistatic packaging until required for installation.**
- **Always transport sensitive components in antistatic packaging.**

Locate jumper blocks J1, J4, J5, and J8, as shown in Figure 1-1. For block J1, make sure that there are jumpers connecting pins 3 and 4 and pins 5 and 6. For blocks J4, J5, and J8 make sure that all positions on each block are jumpered. These jumpers perform the following functions:

- J1 controls the operating mode of the DSP56307.
- J4 and J5 control the interface between the audio codec and the DSP56307 enhanced synchronous serial interface (ESSI0).
- J8 controls the interface between the DSP56307 JTAG/OnCE port and DSP56002 synchronous serial interface (SSI).

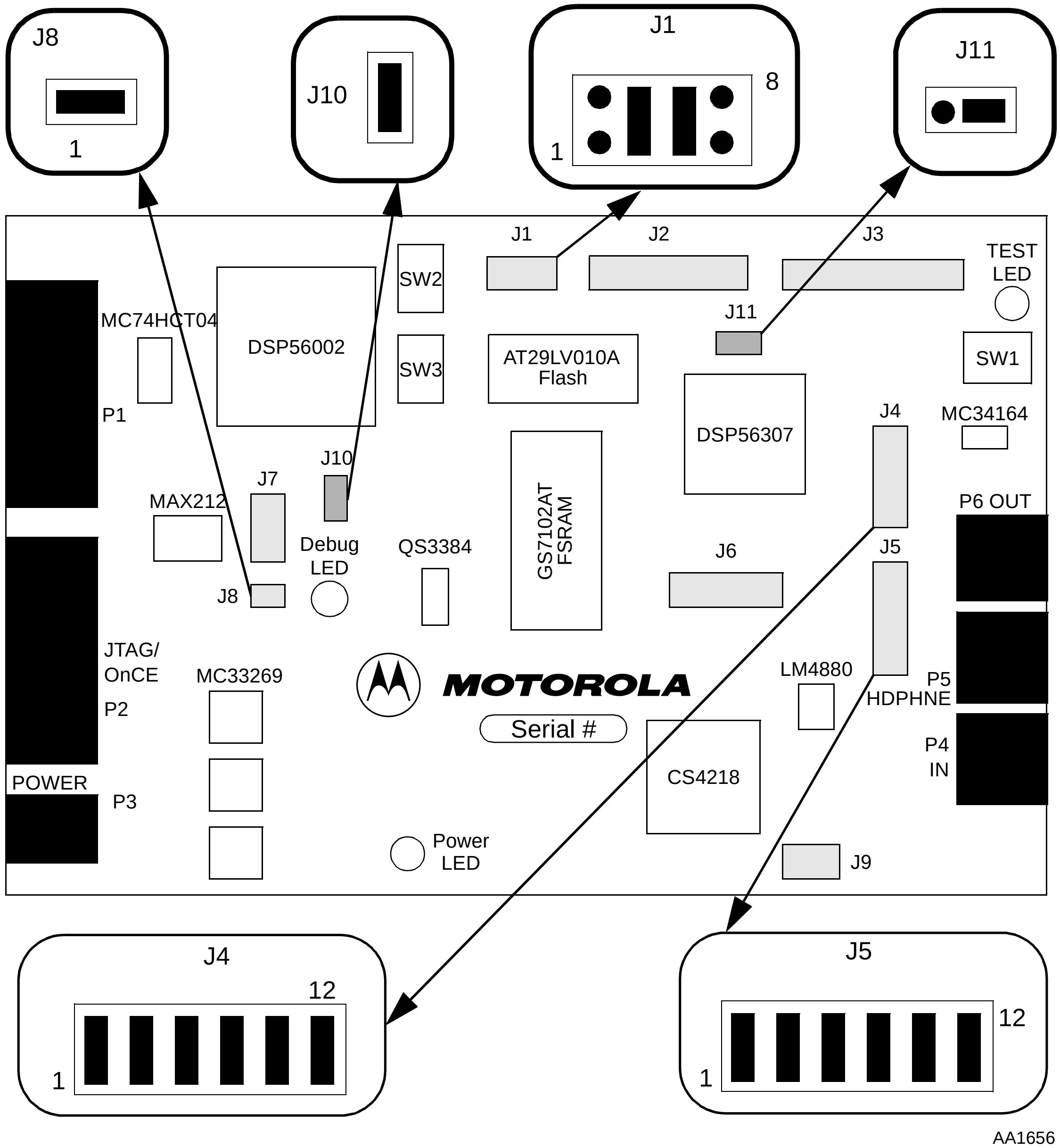


Figure 1-1. DSP56307EVM Component Layout

1.2.2 Connecting the DSP56307EVM to the PC and Power

Figure 1-2 shows the interconnection diagram for connecting the PC and the external power supply to the DSP56307EVM board.

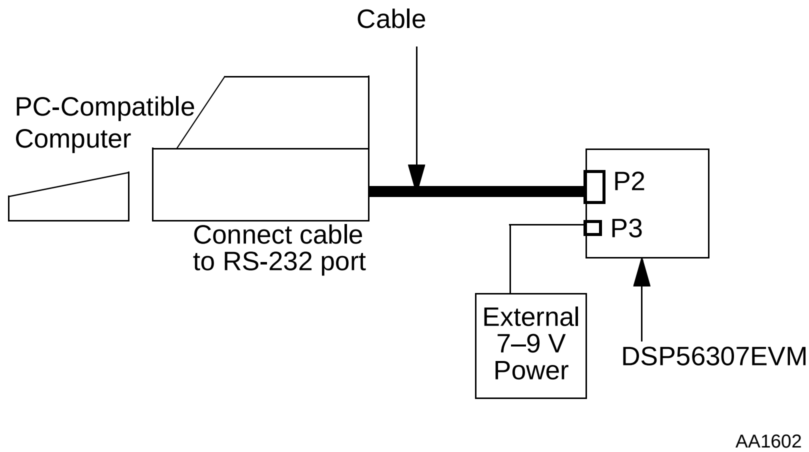


Figure 1-2. Connecting the DSP56307EVM Cables

Use the following steps to complete cable connections:

1. Connect the DB9P end of the RS-232 interface cable to the RS-232 port connection on the PC.
2. Connect the DB9S end of the cable to P2, shown in Figure 1-1, on the DSP56307EVM board. This provides the connection to allow the PC to control the board function.
3. Make sure that the external 7–9 V power supply *does not* have power supplied to it.
4. Connect the 2.1-mm output power plug into P3, shown in Figure 1-1, on the DSP56307EVM board.
5. Apply power to the power supply. The green Power LED lights up when power is correctly applied.

Chapter 2
Example Test Program

This section contains an example that illustrates how to develop a very simple program for the DSP56307. This example is for users with little or no experience with the DSP development tools. The example demonstrates the form of assembly programs, gives instructions on how to assemble programs, and shows how the Debugger can verify the operation of programs.

Figure 2-1 shows the development process flow for assembly programs. The rounded blocks represent the assembly and object files. The white blocks represent software programs to assemble and link the assemble programs. The gray blocks represent hardware products. The following sections give basic information on the assembly program, the assembler, the linker and the object files. For detailed information on these subjects, consult the assembler and linker manuals provided with the Motorola DSP CLAS software package, available through your Motorola sales office or distributor. The documentation is also available through the Motorola Wireless internet URL *http://www.mot.com/SPS/DSP/documentation*.

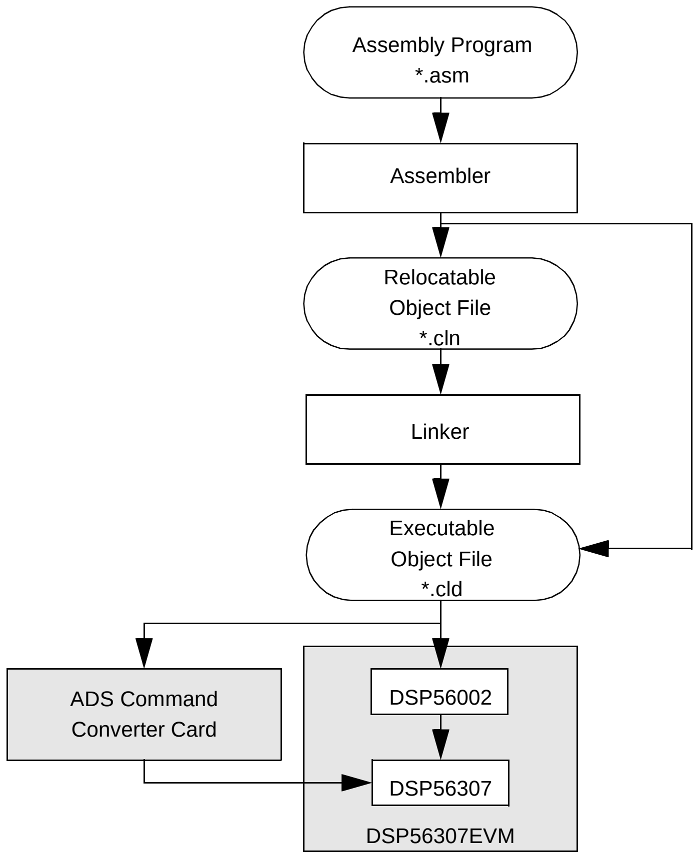


Figure 2-1. Development Process Flow

2.1 Writing the Program

The following sections describe the format of assembly language source statements and give an example assembly program.

2.1.1 Source Statement Format

Programs written in assembly language consist of a sequence of source statements. Each source statement may include up to six fields separated by one or more spaces or tabs: a label field, an operation field, an operand field, up to two data transfer fields, and a comment field. For example, the following source statement shows all six possible fields:

```
trm     mac     x0,y0,a     x:(r0)+,x0      y:(r4)+,y0     ;Text
```

trm	mac	x0,y0,a	x:(r0)+,x0	y:(r4)+,y0	;Text
Label	Operation	Operand	X Data Transfer	Y Data Transfer	Comment

2.1.1.1 Label Field

The label field is the first field of a source statement and can take one of the following forms:

- A space or tab as the first character on a line ordinarily indicates that the label file is empty and that the line has no label.
- An alphabetic character as the first character indicates that the line contains a symbol called a label.
- An underscore as the first character indicates that the label is local.

With the exception of some directives, a label is assigned the value of the location counter of the first word of the instruction or data being assembled. A line consisting of only a label is a valid line and assigns the value of the location counter to the label.

2.1.1.2 Operation Field

The operation field appears after the label field and must be preceded by at least one space or tab. Entries in the operation field may be one of three types:

- **Opcode**—mnemonics that correspond directly to DSP machine instructions
- **Directive**—special operation codes known to the assembler that control the assembly process
- **Macro call—**invocation of a previously–defined macro that is to be inserted in place of the macro call

2.1.1.3 Operand Field

The interpretation of the operand field depends on the contents of the operation field. The operand field, if present, must follow the operation field and must be preceded by at least one space or tab.

2.1.1.4 Data Transfer Fields

Most opcodes specify one or more data transfers to occur during the execution of the instruction. These data transfers are indicated by two addressing mode operands separated by a comma, with no embedded blanks. If two data transfers are specified, they must be separated by one or more blanks or tabs. Refer to the *DSP56300 Family Manual* for a complete discussion of addressing modes that are applicable to data transfer specifications.

2.1.1.5 Comment Field

Comments are not considered significant to the assembler but can be included in the source file for documentation purposes. A comment field is composed of any characters that are preceded by a semicolon.

2.1.2 Example Program

The example program discussed in this section takes two lists of data, one in X memory and one in Y memory, and calculates the sum of the products of the two lists. Calculating the sum of products is the basis for many DSP functions. Therefore, the DSP56307 has a special instruction Multiplier–Accumulate (MAC) which multiplies two values and adds the result to the contents of an accumulator.

Example 2 -1. Simple DSP56307EVM Code Example

```
;**************************************************************
;A SIMPLE PROGRAM: CALCULATING THE SUM OF PRODUCTS
;**************************************************************
PBASE    EQU     $100         ;instruct the assembler to replace
                              ;every occurrence of PBASE with $200
XBASE    EQU     $0           ;used to define the position of the
                              ;data in X memory
YBASE    EQU     $0           ;used to define the position of the
                              ;data in Y memory
;**************************************************************
;X MEMORY
;**************************************************************
         org     x:XBASE      ;instructs the assembler that we
                              ;are referring to X memory starting
                              ;at location XBASE
list1    dc      $475638,$738301,$92673a,$898978,$091271,$f25067
         dc      $987153,$3A8761,$987237,$34b852,$734623,$233763
         dc      $f76756,$423423,$324732,$f40029
;**************************************************************
;Y MEMORY
;**************************************************************
         org     y:YBASE      ;instructs the assembler that we
                              ;are referring to Y memory starting
                              ;at location YBASE
list2    dc      $f98734,$800000,$fedcba,$487327,$957572,$369856
         dc      $247978,$8a3407,$734546,$344787,$938482,$304f82
         dc      $123456,$657784,$567123,$675634
;**************************************************************
;PROGRAM
;**************************************************************
         org     p:0          ;put following program in program
                              ;memory starting at location 0
```

Example 2-1. Simple DSP56307EVM Code Example (Continued)

```
        jmp     begin       ;p:0 is the reset vector i.e. where
                            ;the DSP looks for instructions
                            ;after a reset
        org     p:PBASE     ;start the main program at p:PBASE
begin
        move    #list1,r0   ;set up pointer to start of list1
        move    #list2,r4   ;set up pointer to start of list2
        clr     a           ;clear accumulator a
        move    x:(r0)+,x0     y:(r4)+,y0
                            ;load the value of X memory pointed
                            ;to by the contents of r0 into x0 and
                            ;post-increment r0
                            ;load the value of Y memory pointed
                            ;to by the contents of r4 into y0 and
                            ;post-increment r4
        do      #15,endloop;do 15 times
        mac     x0,y0,a     x:(r0)+,x0    y:(r4)+,y0
                            ;multiply and accumulate, and load
                            ;next values
endloop jmp     *           ;this is equivalent to
                            ;label jmp label
                            ;and is therefore a never-ending,
                            ;empty loop
;*************************************************************
;END OF THE SIMPLE PROGRAM
;*************************************************************
```

2.2 Assembling the Program

The following sections describe the format of the assembler command, list the assembler special characters and directives, and give instructions to assemble the example program.

2.2.1 Assembler Command Format

The Motorola DSP assembler is included with the DSP56307EVM on the Motorola Tools CD and can be installed by following the instructions in the **Read Me** file on the CD. The Motorola DSP assembler is a program that translates assembly language source statements into object programs compatible with the DSP56307. The general format of the command line to invoke the assembler is

asm56300 [options] <filenames>

where ***asm56300*** is the name of the Motorola DSP assembler program, and **<*filenames*>** is a list of the assembly language programs to be assembled.

2.2.2 Assembler Options

Table 2-1 describes the assembler options. To avoid ambiguity, the option arguments should immediately follow the option letter with no blanks between them.

Table 2-1. Assembler Options

Option	Description
-**A**	Puts the assembler into absolute mode and generates an absolute object file when the **-B** command line option is given. By default, the assembler produces a relocatable object file that is subsequently processed by the Motorola DSP linker.
-**B<objfil>**	Specifies that an object file is to be created for assembler output. <objfil> can be any legal operating system filename, including an optional pathname. The type of object file depends on the assembler operation mode. If the **-A** option is supplied on the command line, the assembler operates in absolute mode and generates an absolute object (.cld) file. If there is no **-A** option, the assembler operates in relative mode and creates a relocatable object (.cln) file. If the **-B** option is not specified, the assembler does not generate an object file. If no <objfil> is specified, the assembler uses the basename (filename without extension) of the first filename encountered in the source input file list and appends the appropriate file type (.cln or.cld) to the basename. The **-B** option should be specified only once. Example: ***asm56300 -Bfilter main.asm fft.asm fio.asm*** This example assembles the files main.asm, fft.asm, and fio.asm together to produce the relocatable object file filter.cln.
-**D <symbol> <string>**	Replaces all occurrences of <symbol> with <string> in the source files to be assembled. Example: ***asm56300 -DPOINTS 16 prog.asm*** Replaces all occurrences of the symbol POINTS in the program prog.asm by the string ‘16’.
-EA<errfil> or **-EW<errfil>**	Allows the standard error output file to be reassigned on hosts that do not support error output redirection from the command line. <errfil> must be present as an argument but can be any legal operating system filename, including an optional pathname. The **-EA** option causes the standard error stream to be written to <errfil>; if <errfil> exists, the output stream is appended to the end of the file. The **-EW** option also writes the standard error stream to <errfil>; if <errfil> exists, it is overwritten. Example: ***asm56300 -EWerrors prog.asm*** Redirects the standard output to the file errors. If the file already exists, it is overwritten.
-**F<argfil>**	Indicates that the assembler should read command line input from <argfil>, which can be any legal operation system filename, including an optional pathname. <argfil> is a text file containing further options, arguments, and filenames to be passed to the assembler. The arguments in the file need to be separated only by white space. A semicolon on a line following white space makes the rest of the line a comment. Example: ***asm56300 -Fopts.cmd*** Invokes the assembler and takes the command line options and source filenames from the command file opts.cmd.

Table 2-1. Assembler Options

Option	Description
-**G**	Sends the source file line number information to the object file. This option is valid only in conjunction with the **-B** command line option. Debuggers can use the generated line number information to provide source-level debugging. Example: ***asm56300 -B -Gmyprog.asm*** Assembles the file myprog.asm and sends the source file line number information to the resulting object file myprog.cln.
-**I<pathname>**	Causes the assembler to look in the directory defined by <pathname> for any include file not found in the current directory. <pathname> can be any legal operating system pathname. Example: ***asm56300 -I\project\ testprog*** Uses IBM PC pathname conventions and causes the assembler to prefix any include files not found in the current directory with the \project\ pathname.
-**L<lstfil>**	Specifies that a listing file is to be created for assembler output. <lstfil> can be any legal operating system filename, including an optional pathname. If no <lstfil> is specified, the assembler uses the basename (filename without extension) of the first filename encountered in the source input file list and appends .lst to the basename. The **-L** option is specified only once. Example: ***asm56300 -L filter.asm gauss.asm*** Assembles the files filter.asm and gauss.asm together to produce a listing file. Because no filename is given, the output file is named using the basename of the first source file, in this case filter, and the listing file is called filter.lst.
-M<pathname>	Causes the assembler to look in the directory defined by <pathname> for any macro file not found in the current directory. <pathname> can be any legal operating system pathname. Example: ***asm56300 -Mfftlib\ trans.asm*** Uses IBM PC pathname conventions and causes the assembler to look in the fftlib subdirectory of the current directory for a file with the name of the currently invoked macro found in the source file, trans.asm.
-**V**	Causes the assembler to report assembly progress to the standard error output stream.
-**Z**	Causes the assembler to strip symbol information from the absolute load file. Normally symbol information is retained in the object file for symbolic references purposes. This option is valid only with the **-A** and **-B** options. Note: Multiple options can be used. A typical string might be as follows: Example: ***asm56300 -A -B -L -G filename.asm***

2.2.3 Assembler Directives

In addition to the DSP56307 instruction set, the assembly programs can contain mnemonic directives that specify auxiliary actions to be performed by the assembler. These are the assembler directives. These directives are not always translated into

machine language. The following sections briefly describe the various types of assembler directives.

2.2.3.1 Assembler Significant Characters

The following one-and two-character sequences are significant to the assembler:

;— Comment delimiter

;;— Unreported comment delimiter

\— Line continuation character or macro dummy argument concatenation operator

?— Macro value substitution operator

%— Macro hex value substitution operator

^— Macro local label override operator

“— Macro string delimiter or quoted string DEFINE expansion character

@— Function delimiter

*— Location counter substitution

++— String concatenation operator

[]— Substring delimiter

<<— I/O short addressing mode force operator

<— Short addressing mode force operator

>— Long addressing mode force operator

#— Immediate addressing mode operator

#<— Immediate short addressing mode force operator

#>— Immediate long addressing mode force operator

2.2.3.2 Assembly Control

The directives used for assembly control are as follows:

COMMENT— Start comment lines

DEFINE— Define substitution string

END — End of source program

FAIL— Programmer-generated error message

FORCE— Set operand forcing mode

HIMEM— Set high memory bounds

INCLUDE— Include secondary file

LOMEM— Set low memory bounds

MODE— Change relocation mode

MSG— Programmer-generated message

ORG— Initialize memory space and location counters

RADIX— Change input radix for constants

RDIRECT— Remove directive or mnemonic from table

SCSJMP— Set structured control branching mode

SCSREG— Reassign structured control statement registers

UNDEF— Undefine DEFINE symbol

WARN— Programmer-generated warning

2.2.3.3 Symbol Definition

The directives used to control symbol definition are as follows:

ENDSEC— End section

EQU— Equate symbol to a value

GLOBAL— Global section symbol declaration

GSET— Set global symbol to a value

LOCAL— Local section symbol declaration

SECTION— Start section

SET— Set symbol to a value

XDEF— External section symbol definition

XREF— External section symbol reference

2.2.3.4 Data Definition/Storage Allocation

The directives to control constant data definition and storage allocation are as follows:

BADDR— Set buffer address

BSB— Block storage bit-reverse

BSC— Block storage of constant

BSM— Block storage modulo

BUFFER— Start buffer

DC— Define constant

DCB— Define constant byte

DS— Define storage

DSM— Define modulo storage

DSR— Define reverse carry storage

ENDBUF— End buffer

2.2.3.5 Listing Control and Options

The directives to control the output listing are as follows:

LIST— List the assembly

LSTCOL— Set listing field widths

NOLIST— Stop assembly listing

OPT— Assembler options

PAGE— Top of page/size page

PRCTL— Send control string to printer

STITLE— Initialize program subtitle

TABS— Set listing tab stops

TITLE— Initialize program title

2.2.3.6 Object File Control

The directives for control of the object file are as follows:

COBJ— Comment object code

IDENT— Object code identification record

SYMOBJ— Write symbol information to object file

2.2.3.7 Macros and Conditional Assembly

The directives for macros and conditional assembly are as follows:

DUP— Duplicate sequence of source lines

DUPA— Duplicate sequence with arguments

DUPC— Duplicate sequence with characters

DUPF— Duplicate sequence in loop

ENDIF— End of conditional assembly

ENDM— End of macro definition

EXITM— Exit macro

IF— Conditional assembly directive

MACLIB— Macro library

MACRO— Macro definition

PMACRO— Purge macro definition

2.2.3.8 Structured Programming

The directives for structured programming are as follows:

.BREAK— Exit from structured loop construct

.CONTINUE— Continue next iteration of structured loop

.ELSE— Perform following statements when .IF false

.ENDF— End of .FOR loop

.ENDI— End of .IF condition

.ENDL— End of hardware loop

.ENDW— End of .WHILE loop

.FOR— Begin .FOR loop

.IF— Begin .IF condition

.LOOP— Begin hardware loop

.REPEAT— Begin .REPEAT loop

.UNTIL— End of .REPEAT loop

.WHILE— Begin .WHILE loop

2.2.4 Assembling the Example Program

The assembler is an MS-DOS–based program; thus, to use the assembler you must exit Windows or open an MS-DOS Prompt Window. To assemble the example program, type ***asm56300 -a -b -l -g example.asm*** into the evm30xw directory created by the installation process from Section 2.2.1, "Assembler Command Format," on page 2-5. This creates two additional files: example.cld and example.lst. The example.cld file is the absolute object file of the program; it is downloaded into the DSP56307. The example.lst file is the listing file; it gives full details of where the program and data are placed in the DSP56307 memory.

2.3 Motorola DSP Linker

Though not needed for our simple example, the Motorola DSP linker is also included with the DSP56307EVM. The Motorola DSP linker is a program that processes relocatable object files produced by the Motorola DSP assembler, generating an absolute executable file which can be downloaded to the DSP56307. The Motorola DSP linker is included on the Motorola Tools CD and can be installed by following the instructions in Section 2.2.1, "Assembler Command Format," on page 2-5. The general format of the command line to invoke the linker is

dsplnk [options] <filenames>

where ***dsplnk*** is the name of the Motorola DSP linker program, and **<*filenames*>** is a list of the relocatable object files to be linked.

2.4 Linker Options

Table 2-2 describes the linker options. To avoid ambiguity, the option arguments should immediately follow the option letter with no blanks between them.

Table 2-2. Linker Options

Option	Description
-**A**	Auto-aligns circular buffers. Any modulo or reverse-carry buffers defined in the object file input sections are relocated independently in order to optimize placement in memory. Code and data surrounding the buffer are packed to fill the space formerly occupied by the buffer and any corresponding alignment gaps. Example: ***dsplnk -A myprog.cln*** Links the file myprog.cln and optimally aligns any buffers encountered in the input.
-**B<objfil>**	Specifies that an object file is to be created for linker output. <objfil> can be any legal operating system filename, including an optional pathname. If no filename is specified, or if the **-B** option is not present, the linker uses the basename (filename without extension) of the first filename encountered in the input file list and appends .cld to the basename. If the **-I** option is present (see below), an explicit filename must be given because if the linker follows the default action, it can overwrite one of the input files. The **-B** option is specified only once. If the file named in the **-B** option already exists, it is overwritten. Example: ***dsplnk -Bfilter.cld main.cln fft.cln fio.cln*** Links the files main.cln, fft.cln, and fio.cln together to produce the absolute executable file filter.cld.
-EA<errfil> or **-EW<errfil>**	Allows the standard error output file to be reassigned on hosts that do not support error output redirection from the command line. <errfil> must be present as an argument, but it can be any legal operating system filename, including an optional pathname. The **-EA** option causes the standard error stream to be written to <errfil>; if <errfil> exists, the output stream is appended to the end of the file. The **-EW** option also writes the standard error stream to <errfil>; if <errfil> exists it is overwritten. Example: ***dsplnk -EWerrors myprog.cln*** Redirects the standard error output to the file errors. If the file already exists, it is overwritten.
-**F<argfil>**	Indicates that the linker should read command line input from <argfil>, which can be any legal operating system filename, including an optional pathname. <argfil> is a text file containing further options, arguments, and filenames to be passed to the linker. The arguments in the file need be separated only by white space. A semicolon on a line following white space makes the rest of the line a comment. Example: ***dsplnk -Fopts.cmd*** This example invokes the linker and takes command line options and input filenames from the command file opts.cmd.

Table 2-2. Linker Options

Option	Description
-**G**	Sends source file line number information to the object file. The generated line number information can be used by debuggers to provide source-level debugging. Example: ***dsplnk -B -Gmyprog.cln*** Links the file myprog.cln and sends source file line number information to the resulting object file myprog.cld.
-**I**	The linker ordinarily produces an absolute executable file as output. When the **-I** option is given, the linker combines the input files into a single relocatable object file suitable for reprocessing by the linker. No absolute addresses are assigned and no errors are issued for unresolved external references. Note that the **-B** option must be used when performing incremental linking in order to give an explicit name to the output file. If the filename is allowed to default, it can overwrite an input file. Example: ***dsplnk -I -Bfilter.cln main.cln fft.cln fio.cln*** Combines the files main.cln, fft.cln, and fio.cln to produce the relocatable object file filter.cln.
-**L<library>**	The linker ordinarily processes a list of input files that each contain a single relocatable code module. Upon encountering the **-L** option, the linker treats the following argument as a library file and searches the file for any outstanding unresolved references. If it finds a module in the library that resolves an outstanding external reference, it reads the module from the library and includes it in the object file output. The linker continues to search a library until all external references are resolved or no more references can be satisfied within the current library. The linker searches a library only once, so the position of the **-L** option on the command line is significant. Example: ***dsplnk -B filter main fir -Lio*** Illustrates linking with a library. The files main.cln and fir.cln are combined with any needed modules in the library io.lib to create the file filter.cld.
-**M<mapfil>**	Indicates that a map file is to be created. <mapfil> can be any legal operating system filename, including an optional pathname. If no filename is specified, the linker uses the basename (filename without extension) of the first filename encountered in the input file list and append .map to the basename. If the **-M** option is not specified, then the linker does not generate a map file. The **-M** option is specified only once. If the file named in the **-M** option already exists, it is overwritten. Example: ***dsplnk -M filter.cln gauss.cln*** Links the files filter.cln and gauss.cln to produce a map file. Because no filename is given with the **-M** option, the output file is named using the basename of the first input file, in this case filter. The map file is called filter.map.
-**N**	For the linker the case of symbol names is significant. When the **-**N option is given the linker ignores case in symbol names; all symbols are mapped to lower case. Example: ***dsplnk -N filter.cln fft.cln fio.cln*** Links the files filter.cln, fft.cln, and fio.cln to produce the absolute executable file filetr.cld; Maps all symbol references to lower case.

Table 2-2. Linker Options

Option	Description
-**O<mem>[<ctr>][<map>]:<origin>**	By default, the linker generates instructions and data for the output file beginning at absolute location zero for all DSP memory spaces. This option allows the programmer to redefine the start address for any memory space and associated location counter. <mem> is one of the single-character memory space identifiers (X, Y, L, P). The letter can be upper or lower case. The optional <ctr> is a letter indicating the high (H) or low (L) location counters. If no counter is specified the default counter is used. <map> is also optional and signifies the desired physical mapping for all relocatable code in the given memory space. It can be I for internal memory, E for external memory, R for ROM, A for Port A, and B for Port B. If <map> is not supplied, then no explicit mapping is presumed. The <origin> is a hexadecimal number signifying the new relocation address for the given memory space. The **-O** option can be specified as many times as needed on the command line. This option has no effect if incremental linking is being done. (See the **-I** option.) Example: ***dsplnk -Ope:200 myprog -Lmylib*** Initializes the default P memory counter to hex 200 and maps the program space to external memory.
-**P<pathname>**	When the linker encounters input files, it first searches the current directory (or the directory given in the library specification) for the file. If it is not found and the **-P** option is specified, the linker prefixes the filename (and optional pathname) of the file specification with <pathname> and searches the newly formed directory pathname for the file. The pathname must be a legal operating system pathname. The **-P** option can be repeated as many times as desired. Example: ***dsplnk -P\project\ testprog*** Uses IBM PC pathname conventions and causes the linker to prefix any library files not found in the current directory with the \project\ pathname.
-**R<ctlfil>**	Indicates that a memory control file is to be read to determine the placement of sections into DSP memory and other linker control functions. <ctlfil> can be any legal operating system filename, including an optional pathname. If a pathname is not specified, an attempt is made to open the file in the current directory. If no filename is specified, the linker uses the basename (filename without extension) of the first filename encountered in the link input file list and append .ctl to the basename. If the **-R** option is not specified, then the linker does not use a memory control file. The **-R** option is specified only once. Example: ***dsplnk -Rproj filter.cln gauss.cln*** Links the files filter.cln and gauss.cln using the memory file proj.ctl.
-U<symbol>	Allows the declaration of an unresolved reference from the command line. <symbol> must be specified. This option is useful for creating an undefined external reference in order to force linking entirely from a library. Example: ***dsplnk -Ustart -Lproj.lib*** Declares the symbol start undefined so that it is resolved by code within the library proj.lib.
-V	Causes the linker to report linking progress (beginning of passes, opening and closing of input files) to the standard error output stream. This is useful to insure that link editing is proceeding normally. Example: ***dsplnk -V myprog.cln*** Links the file myprog.cln and sends progress lines to the standard error output.

Table 2-2. Linker Options

Option	Description
-**X<opt>[,<opt>,...,<opt>]**	Provides for link time options that alter the standard operation of the linker. The options are described below. All options can be preceded by NO to reverse their meaning. The **-X<opt>** sequence can be repeated for as many options as desired. **Option Meaning** ABC*Perform address bounds checking AEC*Check form of address expressions ASCEnable absolute section bounds checking CSLCumulate section length data ESODo not allocate memory below ordered sections OVLPWarn on section overlap RO Allow region overlap RSC*Enable relative section bounds checking SVOPreserve object file on errors WEXAdd warning count to exit status (* means default) Example: ***dsplnk -XWEX filter.cln fft.cln fio.cln*** Allows the linker to add the warning count to the exit status so that a project build aborts on warnings as well as errors.
-Z	Allows the linker to strip source file line number and symbol information from the output file. Symbol information normally is retained for debugging purposes. This option has no effect if incremental linking is being done. (See the **-I** option.) Example: ***dsplnk*** -Zfilter.cln fft.cln fio.cln Links the files filter.cln, fft.cln, and fio.cln to produce the absolute object file filter.cln. The output file contains no symbol or line number information.

2.4.1 Linker Directives

Similar to the assembler directives, the linker includes mnemonic directives which specify auxiliary actions to be performed by the linker. Following is a list of the linker directives.

BALIGN—Auto-align circular buffers

BASE— Set region base address

IDENT— Object module identification

INCLUDE— Include directive file

MAP— Map file format control

MEMORY— Set region high memory address

REGION— Establish memory region

RESERVE— Reserve memory block

SBALIGN— Auto-align section buffers

SECSIZE— Pad section length

SECTION— Set section base address

SET— Set symbol value

SIZSYM— Set size symbol

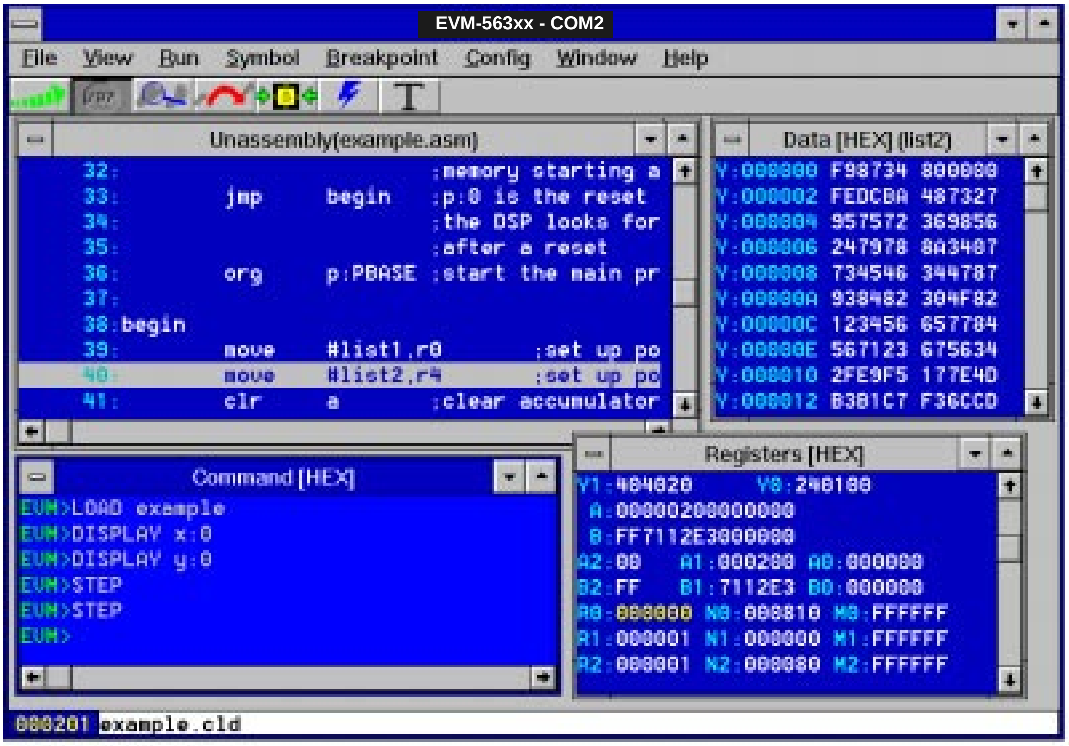
START— Establish start address

SYMBOL— Set symbol value

2.5 Introduction to the Debugger Software

This section briefly introduces the Domain Technologies Debugger, giving only the details required to work through this example. For full details on the Debugger and an informative tutorial, consult the *Debug-56K Manual*. The Domain Technologies Debugger is a software development system for the DSP56307. The Domain Technologies Debugger is included with the DSP56307EVM on the Domain Technologies 3½-inch diskette and can be installed by following the instructions in section 1.4 of the *Debug-56K Manual*. To invoke the Debugger, double-click on the icon labelled *evm30xw* in the EVM5630x program group created when the Debugger was installed.

The Debugger display is similar to that shown in Figure 2-2**;** the screen is divided into four windows—the command window, the data window, the unassembly window, and the registers window. The command window is the window selected, which means that key strokes are placed into the command window. The data window displays DSP56307 data. The unassembly window displays the DSP56307 programs highlighting the next instruction to be executed. The registers window shows the contents of the DSP56307 internal registers.



AA1791

Figure 2-2. Example Debugger Window Display

When the command window is selected as in Figure 2-2, the tool-bar at the top of the screen will change and show buttons for the commands used most often in the command window. From left to right the commands are—go, stop, step, jump, automatic update, reset and radix.

- Go runs the DSP56307 from the program counter.
- Stop stops the DSP56307.
- Step executes a single instruction.
- Jump is similar to the step, except that subroutines are treated as one instruction.
- Automatic update turns the automatic screen update mode on, so that the DSP56307 is periodically interrupted to update the data and registers windows.
- Reset resets the DSP56307.
- Radix can be used to change the radix of the selected window.

Other buttons appear when other windows are selected, and they function as described in the *Debug-56K Manual*.

2.6 Running the Program

To load the example program into the Debugger, click in the command window and type ***load example***. The instruction at line 33 is highlighted in the unassembly window because this is the first instruction to be executed. However, before executing the program, verify that the values expected in data memory are there. To do this, type ***display x:0*** and ***display y:0***. The data is displayed in the data window.

To step through the program, type ***step*** at the command window prompt. For a shortcut, click on the step button or type the start of the command and press the space bar, and the debugger completes the remainder of the command. To repeat the last command, press return. As you step through the code, notice that the registers in the registers window are changed by the instructions. After each cycle, any register that has been changed is highlighted. Once you have stepped through the program, ensure that the program has executed correctly by checking that the result in accumulator a is $FE 9F2051 6DFCC2.

Stepping through the program like this is good for short programs, but it is impractical for large complex programs. The way to debug large programs is to set breakpoints, which are user-defined points where execution of the code stops, allowing the user to step through the section of interest. In the example set a breakpoint, to verify that the values in r0 and r4 are correct before the do loop, type ***break p:$106*** in the command window. The line before the loop brightens in the unassembly window, indicating the breakpoint has been set. To point the DSP56307 back to the start point of the program, type ***change pc 0***. This changes the program counter so that it points to the reset vector. To run the program type ***go*** *or click on the go button*. The DSP56307 stops when it reaches the breakpoint, and you can step through the remainder of the code.

To exit the Debugger, type ***quit*** at the command prompt.

Chapter 3
DSP56307EVM Technical Summary

3.1 DSP56307EVM Description and Features

An overview description of the DSP56307EVM is provided in the DSP56307EVM Product Brief (DSP56307EVMP/D) included with this kit. The main features of the DSP56307EVM include the following:

- DSP56307 24-bit digital signal processor
- FSRAM for expansion memory and Flash PEROM for stand-alone operation.
- 16-bit CD-quality audio codec
- Command converter circuitry

3.2 DSP56307 Description

A full description of the DSP56307, including functionality and user information, is provided in the following documents included as a part of this kit:

- **DSP56307 Technical Data (Document order number DSP56307/D):** Provides features list and specifications including signal descriptions, DC power requirements, AC timing requirements, and available packaging.
- **DSP56307 User's Manual (Document order number DSP56307UM/D):** Provides an overview description of the DSP and detailed information about the on-chip components including the memory and I/O maps, peripheral functionality, and control and status register descriptions for each subsystem.
- **DSP56300 Family Manual (Document order number DSP56300FM/AD):** Provides a detailed description of the core processor including internal status and control registers and a detailed description of the family instruction set.

Refer to these documents for detailed information about chip functionality and operation.

Note: A detailed list of known chip errata is also provided with this kit. Refer to the *DSP56307 Chip Errata* document for information that has changed since the publication of the reference documentation listed above. The latest version can be obtained on the Motorola DSP worldwide web site at *http://www.mot.com/SPS/DSP/chiperrata/index.html*

The core voltage of the DSP56307 can be selected to be +2.5 V_{DC} or +3.3V_{DC} by using jumper J11. Refer to Table 3-1 for jumper options.

Table 3-1. J11 Jumper Options

J11	Voltage Selected
1—2	+3.3 V
2—3	+2.5 V

3.3 Memory

The DSP56307EVM includes the following external memory:

- 64K × 24-bit fast static RAM (FSRAM) for expansion memory
- 128K × 8-bit flash memory for stand-alone operation

Refer to Figure 3-1 for the location of the FSRAM and Flash on the DSP56307EVM. Figure 3-2 shows a functional block diagram of the DSP56307EVM including the memory devices.

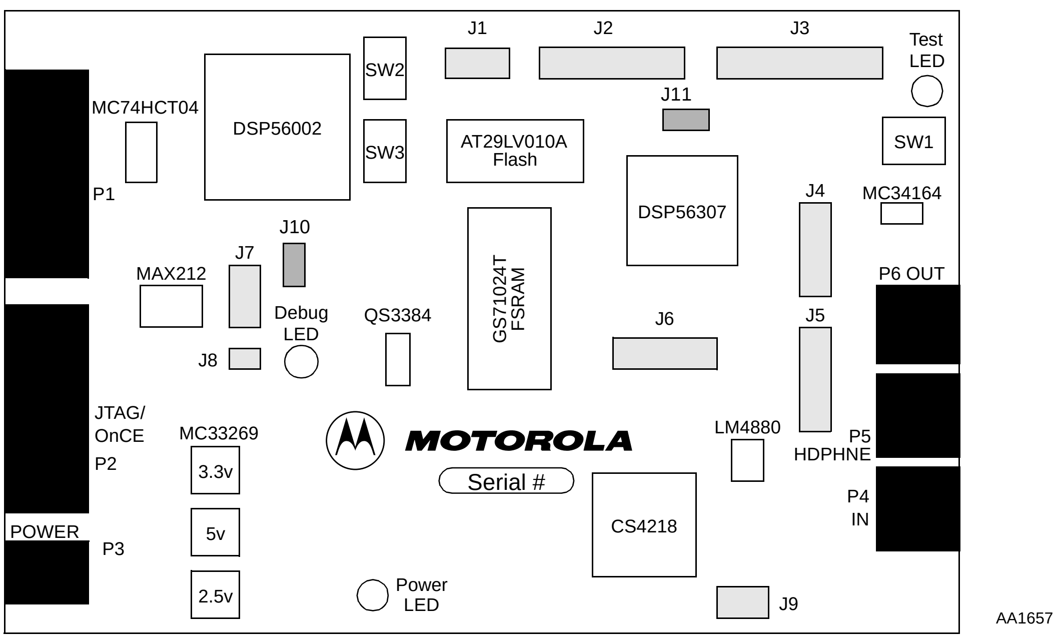


Figure 3-1. DSP56307EVM Component Layout

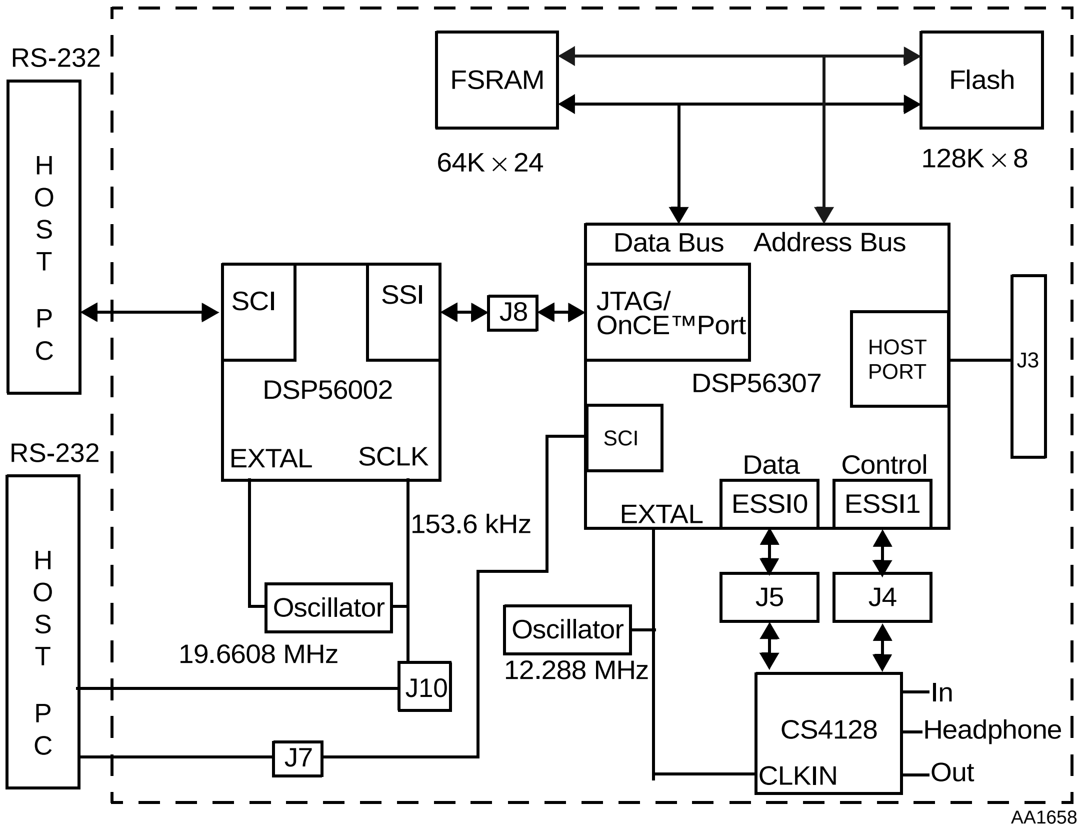


Figure 3-2. DSP56307EVM Functional Block Diagram

3.3.1 FSRAM

The DSP56307EVM uses one bank of 64K × 24-bit fast static RAM(GS71024T-10, labelled U4) for memory expansion. The GS71024T-10 uses a single 3.3 V power supply and has an access time of 10 ns. The following sections detail the operation of the FSRAM.

3.3.1.1 FSRAM Connections

The basic connection for the FSRAM is shown in Figure 3-3.

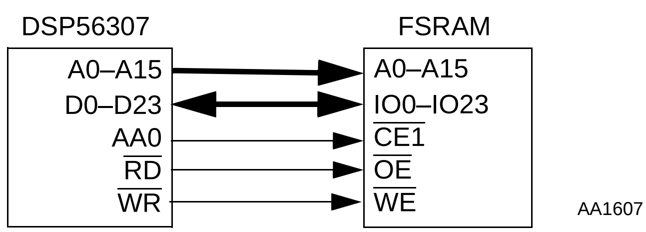


Figure 3-3. FSRAM Connections to the DSP56307

The data input/output pins IO0–IO23 for the FSRAM are connected to the DSP56307 D0–D23 pins. The FSRAM write ($\overline{\text{WE}}$) and output enable ($\overline{\text{OE}}$) lines are connected to the DSP56307 write ($\overline{\text{WR}}$) and read ($\overline{\text{RD}}$) lines, respectively. The FSRAM chip enable ($\overline{\text{CE1}}$) is generated by the DSP56307 address attribute 0 (AA0). The FSRAM activity is controlled by AA0 and the corresponding address attribute register 0 (AAR0). The FSRAM address input pins, A0–A15, are connected to the respective port A address pins of the DSP. This configuration selects a unified memory map of 64K words. The unified memory does not contain partitioned X data, Y data, and program memory. Thus, access to P:$1000, X:$1000, and Y:$1000 are treated as accesses to the same memory cell and 48-bit long memory data moves are not possible to or from the external FSRAM.

3.3.1.2 Example: Programming AAR0

As mentioned above, the FSRAM activity is controlled by the DSP56307 pin AA0 and the corresponding AAR0. AAR0 controls the external access type, the memory type, and which external memory addresses access the FSRAM. Figure 3-4 shows the memory map that is attained with the AAR0 settings described in this example.

Note: In this example, the memory switch bit in the operating mode register (OMR) is cleared and the 16-bit compatibility bit in the status register is cleared.

In Figure 3-4, the FSRAM responds to the 64K of X and Y data memory addresses between $040000 and $04FFFF. However, with the unified memory map, accesses to the same external memory location are treated as accesses to the same memory cell.

A priority mechanism exists among the four AAR control registers. AAR3 has the highest priority and AAR0 had the lowest. Bit 14 of the OMR, the address priority disable (APD) bit, controls which AA pins are asserted when a selection conflict occurs (i.e. the external address matches the address and the space that is specified in more than one AAR). If the APD bit is cleared when a selection conflict occurs, only the highest priority AA pin is asserted. If the APD bit is set when a selection conflict occurs, the lower priority AA pins are asserted in addition to the higher priority AA pin. For this example, only one AA pin must be asserted, AA0. Thus, the APD bit can be cleared.

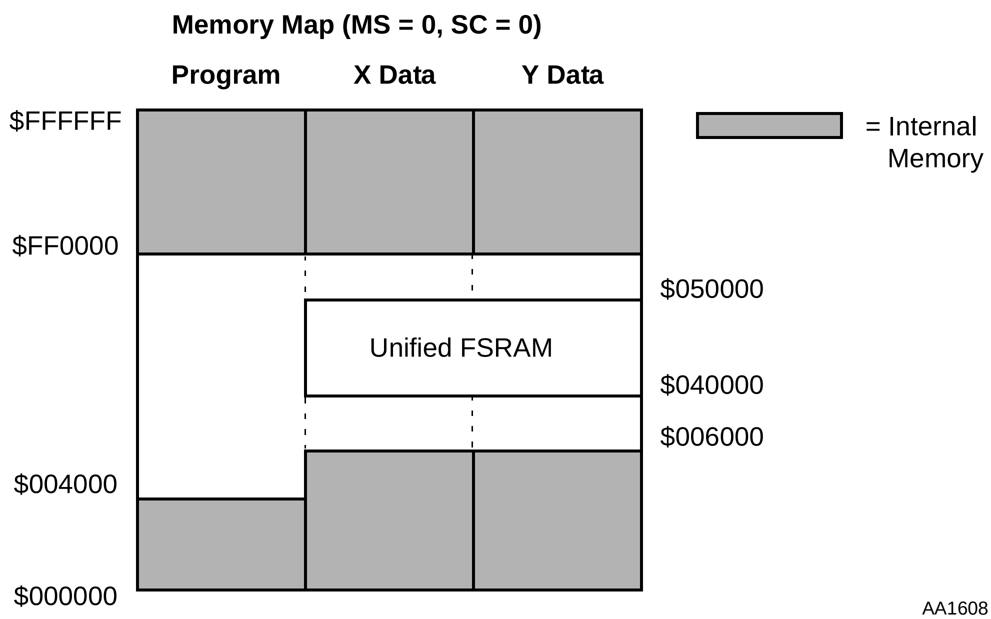


Figure 3-4. Example Memory Map with the Unified External Memory

Figure 3-5 shows the settings of AAR0 for this example. The external access type bits (BAT1 and BAT0) are set to 0 and 1, respectively, to denote FSRAM access. The address attribute polarity bit (BAAP) is cleared to define AA0 as active low. Address multiplexing is not needed with the FSRAM; therefore, the address multiplexing bit BAM is cleared. Packing is not needed with the FSRAM; thus, the packing enable bit BPAC is cleared to disable this option.

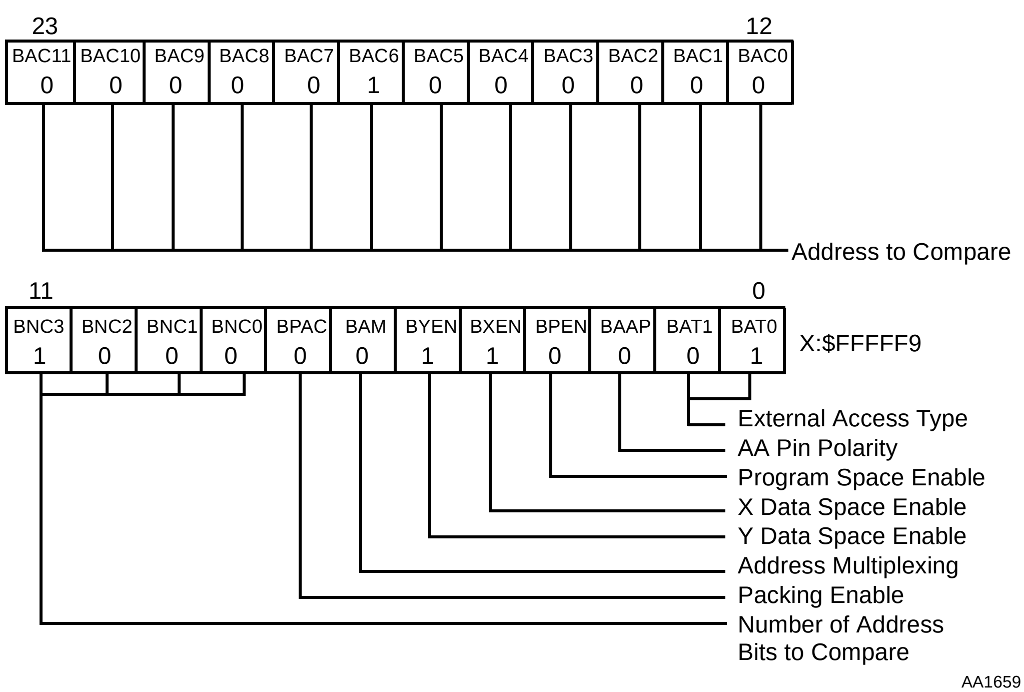


Figure 3-5. Address Attribute Register AAR0

The P, X data, and Y data space Enable bits (BPEN, BXEN, and BYEN) define whether the FSRAM is activated during external P, X data, or Y data space accesses, respectively. For this example, the BXEN and BYEN bits are set, and BPEN is cleared to allow the FSRAM to respond to X and Y data memory accesses only.

The number of address bits to compare BNC(3:0) and the address to compare bits BAC(11:0) determine which external memory addresses access the FSRAM.   The BNC bits define the number of upper address bits that are compared between the BAC bits and the external address to determine if the FSRAM is accessed. For this example, the FSRAM is assigned to respond to addresses between $040000 and $04FFFF. Thus, the BNC bits are set to $8 and the BAC bits are set to $040. If the eight most significant bits of the external address are 00000100, the FSRAM is accessed.

3.3.2 Flash

The DSP56307EVM uses an Atmel AT29LV010A-20TC chip (U3) to provide a 128K× 8-bit CMOS Flash for stand-alone operation (i.e., startup boot operation without accessing the DSP56307 through the JTAG/OnCE port). The AT29LV010 uses a 3.3 V power supply and has a read access time of 200 ns.

3.3.2.1 Flash Connections

The basic connection for the Flash is shown in Figure 3-6.

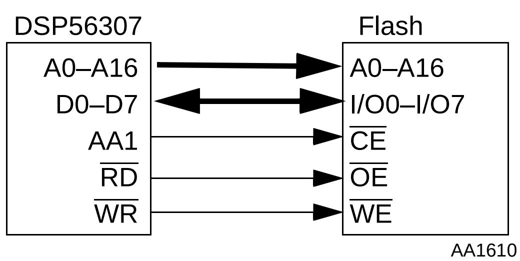


Figure 3-6.   Flash Connections

The flash address pins (A0–A16) connect the respective port A address pins on the DSP. The flash data input/output pins I/O0–I/O7 are connected to the DSP56307 D0–D7 pins. The flash write enable ($\overline{WE}$) and output enable ($\overline{OE}$) lines connect the DSP56307 write ($\overline{WR}$) and read ($\overline{RD}$) enable lines, respectively. Address attribute 1 (AA1) generates the flash chip enable $\overline{CE}$.

3.3.2.2 Programming for Stand-Alone Operation

The DSP56307 mode pins determine the chip operating mode and start-up procedure when the DSP56307 exits the reset state. The switch at SW1 resets the DSP56307 by

asserting and then clearing the $\overline{\text{RESET}}$ pin of the DSP56307. The mode pins MODA, MODB, MODC, and MODD are sampled as the DSP56307 exits the reset state. The mode pins for the DSP56307EVM are controlled by jumper block J1 shown in Figure 3-1 on page 3-2 and Table 3-15 on page 3-16. The DSP56307 boots from the Flash after reset if there are jumpers connecting pins 3 and 4 and pins 5 and 6 on J1 (Mode 1: MODA and MODD are set, and MODB and MODC are cleared).

3.4 Audio Codec

The DSP56307EVM analog section uses Crystal Semiconductor's CS4218-KQ for two channels of 16-bit A/D conversion and two channels of 16-bit D/A conversion. Refer to Figure 3-1 on page 3-2 for the location of the codec on the DSP56307EVM and to Figure 3-2 on page 3-3 for a functional diagram of the codec within the evaluation module. The CS4218 uses a 3.3 V digital power supply and a 5 V analog power supply.

The CS4218 is driven by a 12.288 MHz signal at the codec Master Clock (CLKIN) input pin. The oscillator at Y1 creates a 5 V 12.288 MHz signal. The QS3384 at U5 then converts the 5 V signal to 3.3 V for input to the codec CLKIN pin and the DSP56307 EXTAL pin. Refer to the CS4218 data sheet included with this kit for more information.

The CS4218 is very flexible, offering selectable sampling frequencies between 8 kHz and 48 kHz. The sampling frequency is selected using jumpers on jumper block J9. Table 3-2 shows jumper positions that select the possible sampling frequencies for the DSP56307EVM.

Table 3-2. CS4218 Sampling Frequency Selection

J9 Pins 1–2 (MF6)	J9 Pins 3–4 (MF7)	J9 Pins 5–6 (MF8)	Sampling Frequency (kHz)
Jumper	Jumper	Jumper	48.0
Jumper	Jumper	Open	32.0
Jumper	Open	Jumper	24.0
Jumper	Open	Open	19.2
Open	Jumper	Jumper	16.0
Open	Jumper	Open	12.0
Open	Open	Jumper	9.6
Open	Open	Open	8

The codec is connected to the DSP56307 ESSI0 through the shorting jumpers on J4 and J5 shown in Figure 3-1 on page 3-2. Jumper block J4 connects the ESSI1 pins of the DSP56307 to the control pins of the CS4218. Jumper block J5 connects the ESSI0 pins of the DSP56307 to the data pins of the CS4218. By removing these jumpers, the user has full access to the ESSI0 and ESSI1 pins of the DSP56307. The following sections describe the connections for the analog and digital sections of the codec.

3.4.1 Codec Analog Input/Output

The DSP56307EVM contains 1/8-inch stereo jacks for stereo input, output, and headphones. Figure 3-7 shows the analog circuitry of the codec.

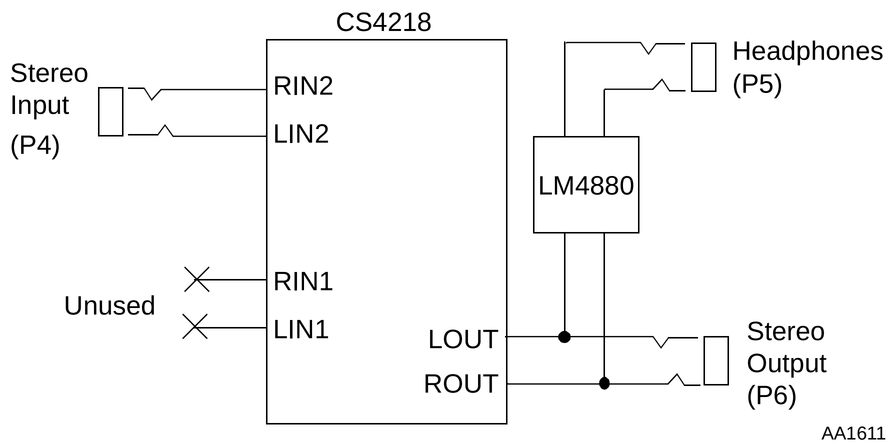


Figure 3-7. Codec Analog Input/Output Diagram

The stereo jack labelled P4/IN on the DSP56307EVM connects to the codec right and left input pins, RIN2 and LIN2. Standard line level inputs are 2 V_{PP} and the codec requires that input levels be limited to 1 V_{PP}. Thus, a voltage divider forms a 6 dB attenuator between P4 and the CS4218.

The codec right and left channel output pins, ROUT and LOUT, provide their output analog signals, through the stereo jack labelled P6/OUT on the DSP56307EVM. The outputs of the codec are also connected to the stereo jack labelled P5/HDPHNE on the DSP56307EVM through National Semiconductor’s LM4880 dual audio power amplifier at U8. The headphone stereo jack permits direct connection of stereo headphones to the DSP56307EVM.

3.4.2 Codec Digital Interface

Figure 3-8 shows the digital interface to the codec. Table 3-3 and Table 3-4 show the jumper selections to Enable/Disable the code’s digital signals.

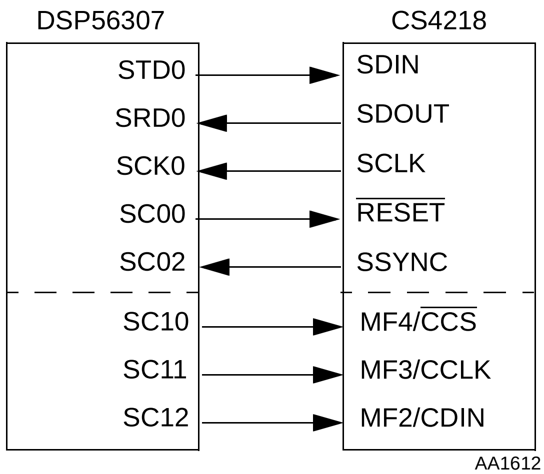


Figure 3-8. Codec Digital Interface Connections

Table 3-3. JP5 Jumper Block Options

JP5	DSP Signal Name	Code Signal Name
1—2	SCK0	SCLK
3—4	SC00	$\overline{\text{RESET}}$
5—6	STD0	SDIN
7—8	SRD0	SDOUT
9—10	SC01	—
11—12	SC02	SSYNC

Table 3-4. JP4 Jumper Block Options

JP4	DSP Signal Name	Code Signal Name
1—2	SCK1	—
3—4	SC10	$\overline{\text{CCS}}$
5—6	STD1	—
7—8	SRD1	—
9—10	SC12	CDIN
11—12	SC11	CCLK

The serial interface of the codec transfers digital audio data and control data into and out of the device. The codec communicates with the DSP56307 through the ESSI0 for the data

information and through the ESSI1 for the control information. The codec has three modes of serial operation that are selected by the serial mode select SMODE1, SMODE2, and SMODE3 pins. The SMODE pins on the DSP56307EVM are set to enable serial mode 4, which separates the audio data from the control data. The SMODE pins are also set to enable the master sub-mode with 32-bit frames, the first 16 bits being the left channel, and the second 16 bits being the right channel.

The DSP56307 ESSI0 transfers the data information to and from the codec. The DSP56307 serial transmit data (STD0) pin transmits data to the codec. The DSP56307 serial receive data (SRD0) pin receives data from the codec. These two pins are connected to the codec serial port data in (SDIN) and serial port data out (SDOUT) pins, respectively. In master sub-mode, the codec serial port clock (SCLK) pin provides the serial bit rate clock for the ESSI0 interface. It is connected to the DSP56307 bidirectional serial clock (SCK0) pin. The DSP56307 serial control 0 (SC00) pin is programmed to control the codec reset signal $\overline{\text{RESET}}$. The serial control 2 (SC02) pin is connected to the codec serial port sync signal (SSYNC) signal. A rising edge on SSYNC indicates that a new frame is about to start.

The DSP56307 ESSI1 pins are used as general purpose i/o (GPIO) signals to transfer the control data to the codec. The control data needs to be transferred only when it changes. The DSP56307 serial control 0 (SC10) pin is programmed to control the codec multi-function pin 4 or the control data chip select pin, MF4/$\overline{\text{CCS}}$. This pin must be low for entering control data. The serial control 1 (SC11) pin connects to the codec multi-function pin 3 or the control data clock pin, MF3/CCLK. The control data is inputted on the rising edge of CCLK. The serial control 2 (SC12) pin is connected to the codec multi-function pin 2 or the control data input pin, MF2/CDIN. This pin contains the control data for the codec.

3.5 Command Converter

The DSP56307EVM uses Motorola’s DSP56002 to perform JTAG/OnCE command conversion. The DSP56002 serial communications interface (SCI) communicates with the host PC through an RS-232 connector. The DSP56002 SCI receives commands from the host PC. The set of commands may include read data, write data, reset OnCE module, reset DSP56307 (the HA2 pin of the DSP56002 is then used to reset the DSP56307), request OnCE module, or release OnCE module. The DSP56002 command converter software interprets the commands received from the PC and sends a sequence of instructions to the DSP56307’s JTAG/OnCE port. The DSP56307 may then continue to receive data or it may transmit data back to the DSP56002. The DSP56002 sends a reply to the host PC to give status information. The set of replies may include acknowledge

good, acknowledge bad, in debug mode, out of debug mode, or data read. When the DSP56307 is in the debug state, the red debug LED (LED2) is illuminated.

The DSP56002 connects to the DSP56307 JTAG/OnCE port through the shorting jumper on J8. Table 3-5 shows the JTAG enable/disable options. The jumper must be present in J8 to use the DSP56002 as the command converter. Refer to Figure 3-1 on page 3-2 for the location of J8 on the DSP56307EVM and to Figure 3-2 on page 3-3 for a functional diagram. Figure 3-9 shows the RS-232 serial interface diagram. Table 3-6 shows the RS-232 connectors pinout, (P2).

Table 3-5. On-Board JTAG Enable/Disable Option

J8	Option Selected
1–2	On–Board Command Converter Enabled
OPEN	On–Board Command Converter Disabled

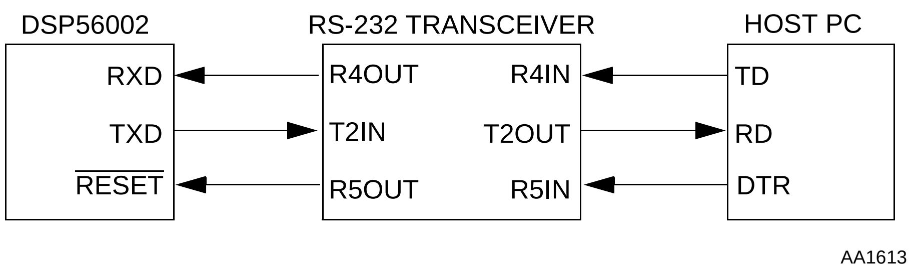


Figure 3-9. RS-232 Serial Interface

Table 3-6. Debug RS-232 Connector (P2) Pinout

Pin Number	DSP Signal Name	Pin Number	DSP Signal Name
1	—	6	—
2	TxD	7	—
3	RxD	8	—
4	$\overline{\text{RESET}}$	9	—
5	GND		

Maxim's 3 V Powered RS-232 Transceiver MAX212 at U11 is used to transmit the signals between the host PC and the DSP56002. Serial data is transmitted from the host PC transmitted data (TD) signal and received on the DSP56002 receive data (RXD) pin.

Serial data is similarly transmitted from the DSP56002 transmit data (TXD) signal and received on the host PC received data (RD) signal. The data terminal ready (DTR) pin asserts the $\overline{\text{RESET}}$ pin of the DSP56002.

As an option, the DSP56307EVM 14-pin JTAG/OnCE connector at J6 allows the user to connect an ADS command converter card directly to the DSP56307EVM, if the DSP56002 command converter software is not used (J8 jumper removed). Pin 8 has been removed from J6 so that the cable cannot be connected to the DSP56307EVM incorrectly. Table 3-7 shows the JTAG/OnCE (J6) connector pinout. The JTAG cable from the ADS command converter is similarly keyed so that the cable cannot be connected to the DSP56307EVM incorrectly.

Table 3-7. JTAG/OnCE (J6) Connector Pinout

Pin Number	DSP Signal Name	Pin Number	DSP Signal Name
1	TDI	2	GND
3	TDO	4	GND
5	TCK	6	GND
7	—	8	KEY-PIN
9	$\overline{\text{PRESET}}$	10	TMS
11	+3.3 V	12	—
13	$\overline{\text{DEZ}}$	14	$\overline{\text{TRST}}$

3.6 Off-Board Interfaces

The DSP56307EVM provides interfaces with off-board devices via its on-chip peripheral ports. Most of the DSP ports are connected to headers on the EVM to facilitate direct access to these pins by using connectors or jumpers.

3.6.1 Serial Communication Interface Port (SCI)

Connection to the DSP’s SCI port can be made at J7. Refer to Table 3-8 for pinout. The signals at J7 are +3.3 V signals straight from the DSP. If RS-232 level signals are required, jumpers should be installed at J7. Refer to Table 3-9 to route the DSP’s SCI signals through an RS-232 level converter to P1. The pinout of P1 is shown in Table 3-10.

By installing a jumper at J10, the SCI port will be clocked by the on-board 153.6 kHz oscillator instead of being clocked externally via the serial port connector, P1, or internally by an SCI timer. If J10 is installed, jumper 3–4 on J7 must be removed.

Table 3-8. SCI Header (J7) Pinout

Pin Number	DSP Signal Name	Pin Number	DSP Signal Name
1	RxD	2	—
3	SCLK	4	—
5	TxD	6	—

Table 3-9. J7 Jumper Options

J7	DSP Signal Name
1—2	RxD
3—4	SCLK
5—6	TxD

Table 3-10. DSP Serial Port (P1) Connector Pinout

Pin Number	DSP Signal Name	Pin Number	DSP Signal Name
1	—	6	—
2	TxD	7	SCLK
3	RxD	8	—
4	$\overline{\text{RESET}}$	9	—
5	GND		

3.6.2 Enhanced Synchronous Serial Port 0 (ESSI0)

Connection to the DSP’s ESSI0 port can be made at J5. Refer to Table 3-11 for the header’s pinout.

Table 3-11. ESSI0 Header (J5) Pinout

Pin Number	DSP Signal Name	Pin Number	DSP Signal Name
1	SCK0	2	—
3	SC00	4	—
5	STD0	6	—
7	SRD0	8	—
9	SC01	10	—
11	SC02	12	—

3.6.3 Enhanced Synchronous Serial Port 1 (ESSI1)

Connection to the DSP's ESSI1 port can be made at J4. Refer to Table 3-12 for the header's pinout.

Table 3-12. ESSI0 Header (J4) Pinout

Pin Number	DSP Signal Name	Pin Number	DSP Signal Name
1	SCK1	2	—
3	SC10	4	—
5	STD1	6	—
7	SRD1	8	—
9	SC12	10	—
11	SC11	12	—

3.6.4 Host Port (HI08)

Connection to the DSP's HI08 port can be made at J3. Refer to Table 3-13 for the header's pinout.

Table 3-13. HI08 Header (J3) Pinout

Pin Number	DSP Signal Name	Pin Number	DSP Signal Name
1	H0	2	H1
3	H2	4	H3
5	H4	6	GND
7	H5	8	H6
9	H7	10	$\overline{\text{RESET}}$
11	HA0	12	HA1
13	HA2	14	$\overline{\text{HCS}}$
15	$\overline{\text{HREQ}}$	16	$\overline{\text{HDS}}$
17	+3.3 V	18	$\overline{\text{HACK}}$
19	HRW	20	GND

3.6.5 Expansion Bus Control

Connection to the DSP’s expansion BUS control signals can be made at J2. Refer to Table 3-14 for header’s pinout.

Table 3-14. Expansion Bus Control Signal Header (J2) Pinout

Pin Number	DSP Signal Name	Pin Number	DSP Signal Name
1	+3.3 V	2	$\overline{\text{RD}}$
3	$\overline{\text{WR}}$	4	$\overline{\text{BG}}$
5	$\overline{\text{BB}}$	6	$\overline{\text{BR}}$
7	$\overline{\text{TA}}$	8	BCLK
9	$\overline{\text{BCLK}}$	10	$\overline{\text{CAS}}$
11	CLKOUT	12	AA1
13	AA0	14	AA2
15	AA3	16	GND

3.7 Mode Selector

Boot Up mode selection for the DSP56307 is made by jumper selections on header J1. Refer to Table 3-15 for header J1 jumper options.

Table 3-15. Boot Mode Selection Options

Mode Number	J1				Boot Mode Selected
	D 1-2	C 3-4	B 5-6	A 7-8	
8	OPEN	JUMP	JUMP	JUMP	Jump to program at $008000
9	OPEN	JUMP	JUMP	OPEN	Bootstrap from byte-wide memory
10	OPEN	JUMP	OPEN	JUMP	Bootstrap from SCI
12	OPEN	OPEN	JUMP	JUMP	HI08 bootstrap in ISA/DSP5630X mode
13	OPEN	OPEN	JUMP	OPEN	HI08 Bootstrap in HC11 non-multiplexed bus mode
14	OPEN	OPEN	OPEN	JUMP	HI08 Bootstrap in 8051 multiplexed bus mode.
15	OPEN	OPEN	OPEN	OPEN	HI08 Bootstrap in MC68302 bus mode.

Appendix A
DSP56307EVM Schematics

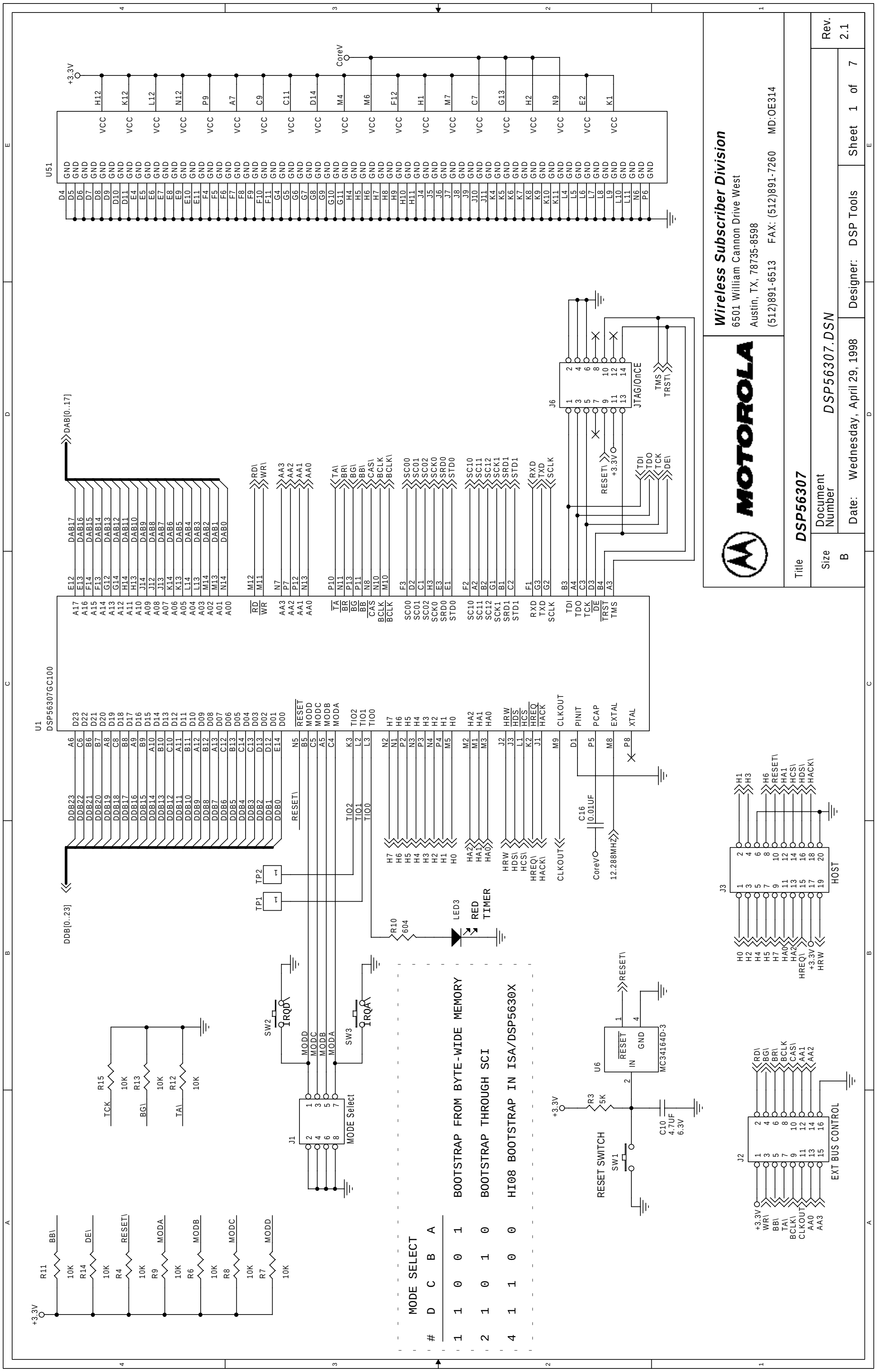


Figure A-1. DSP56307

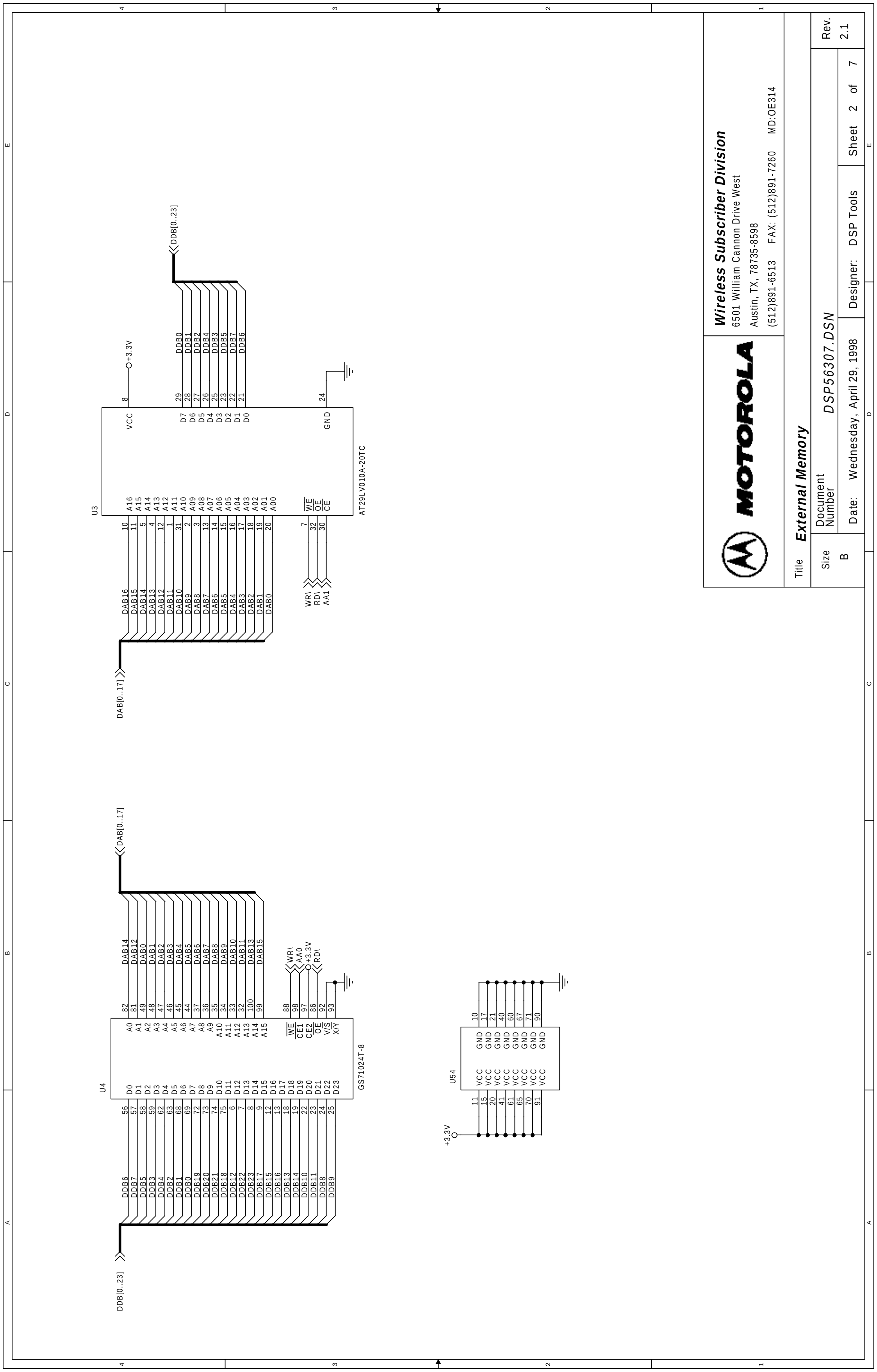


Figure A-2. External Memory

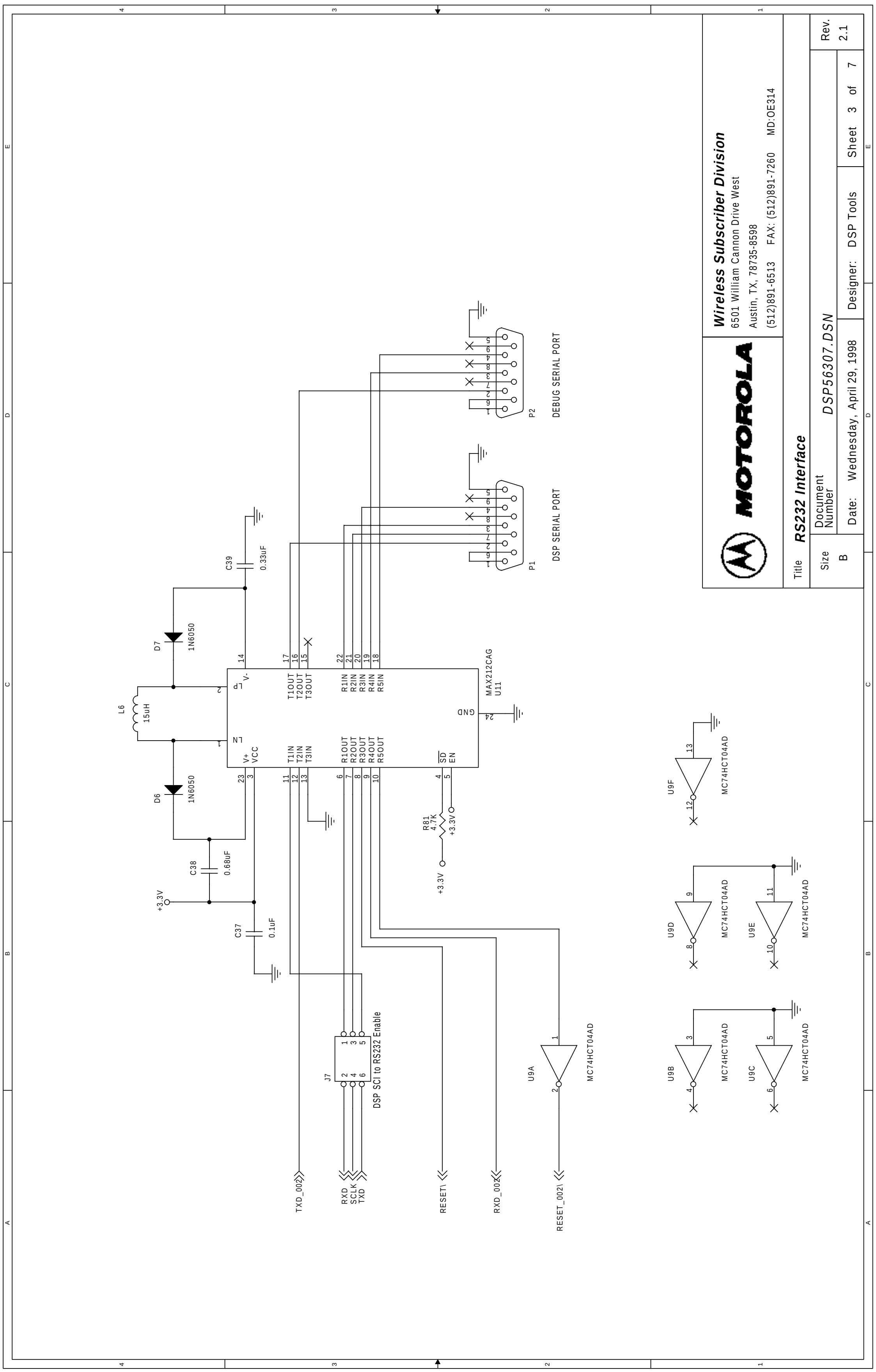


Figure A-3. RS232 Interface

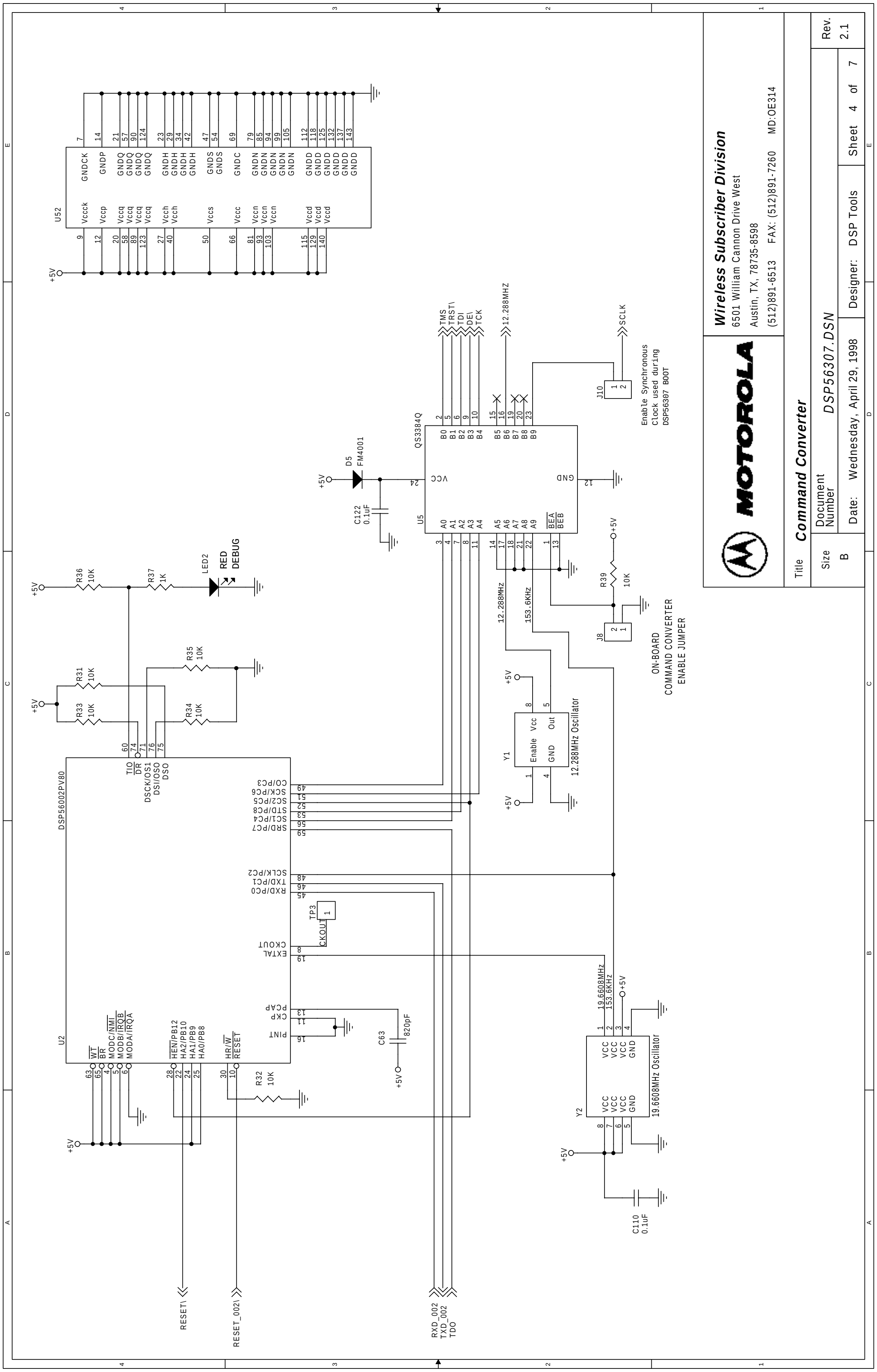


Figure A-4. Command Converter

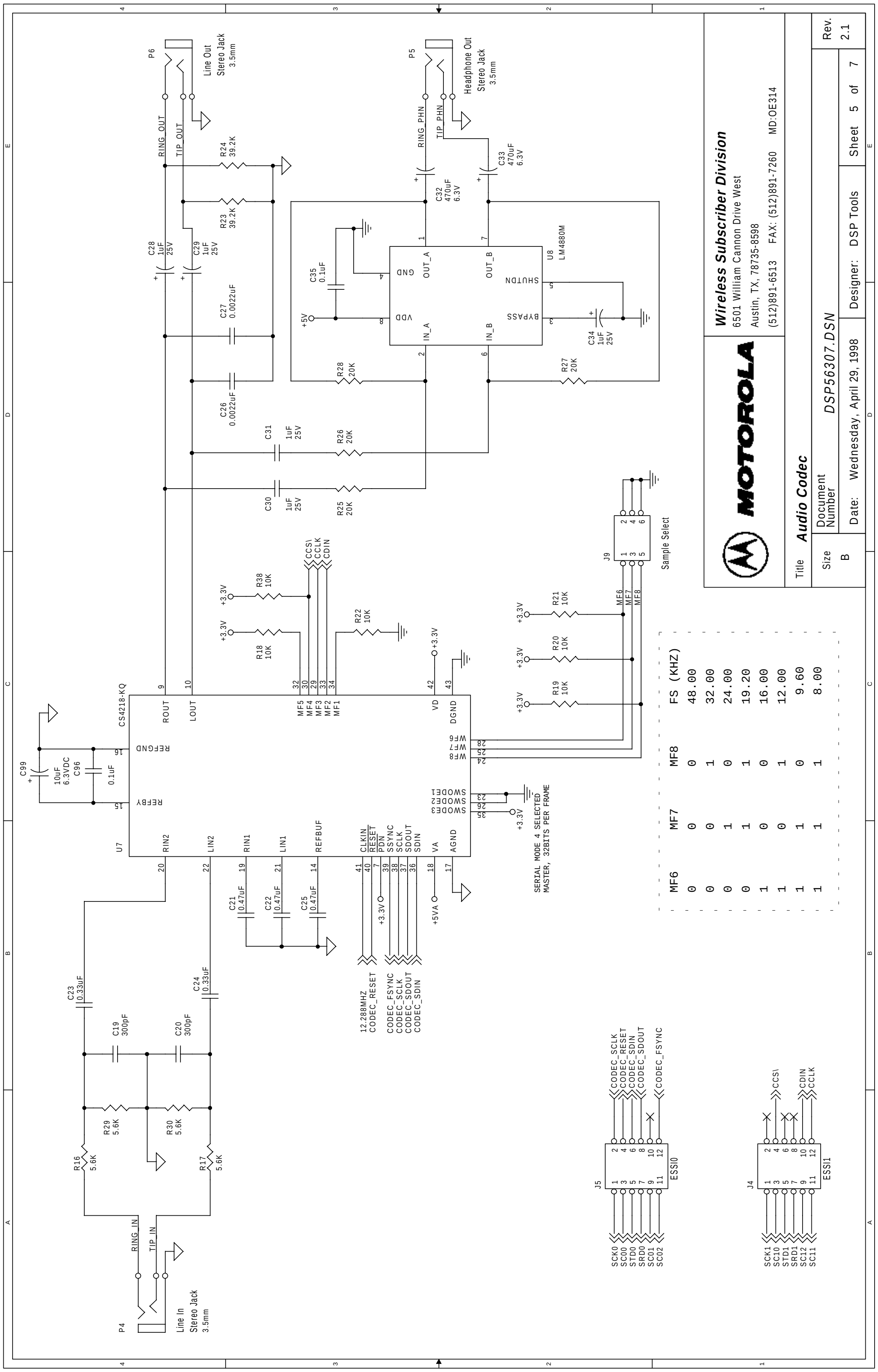


Figure A-5. Audio Codec

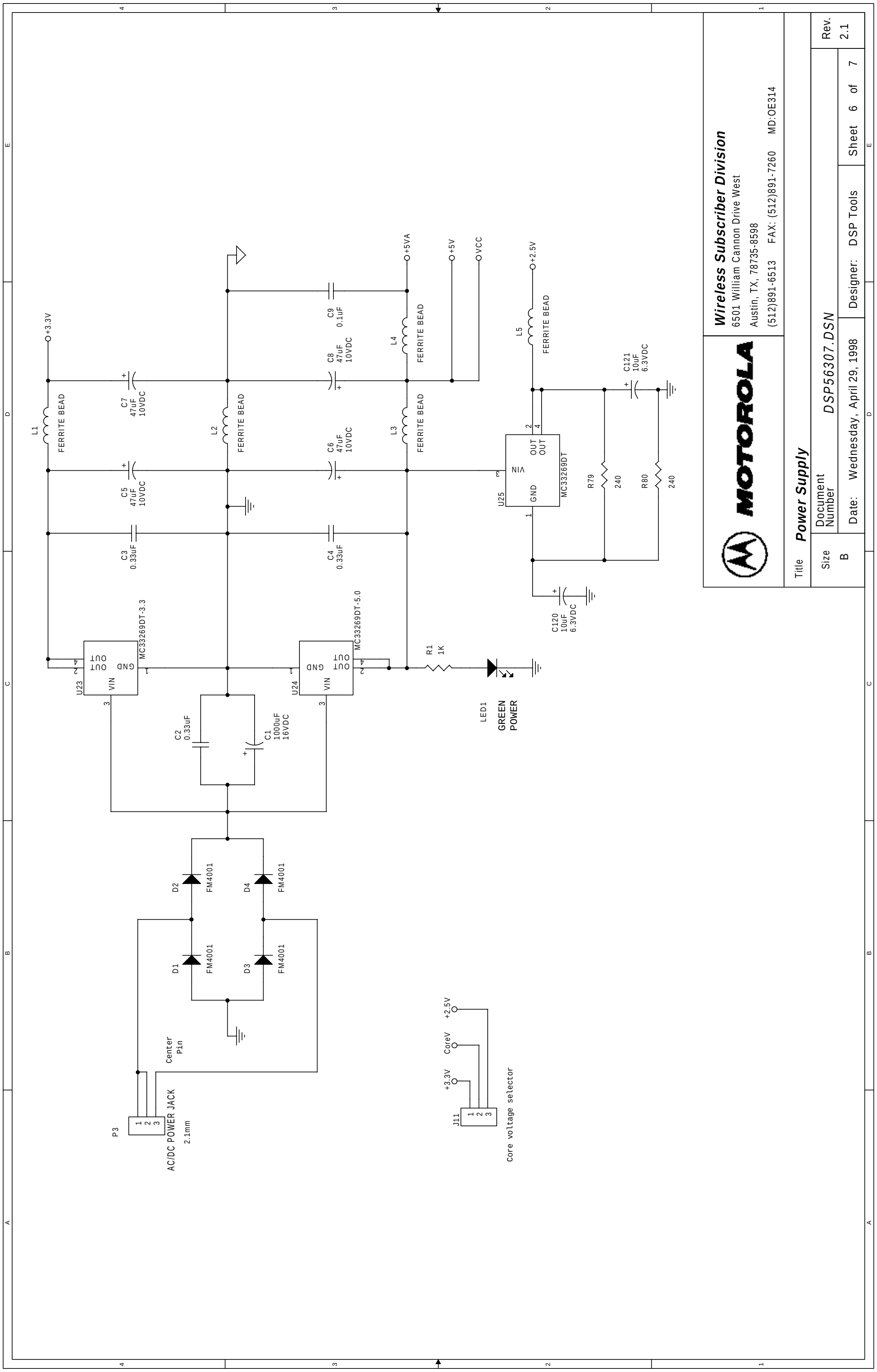


Figure A-6. Power Supply

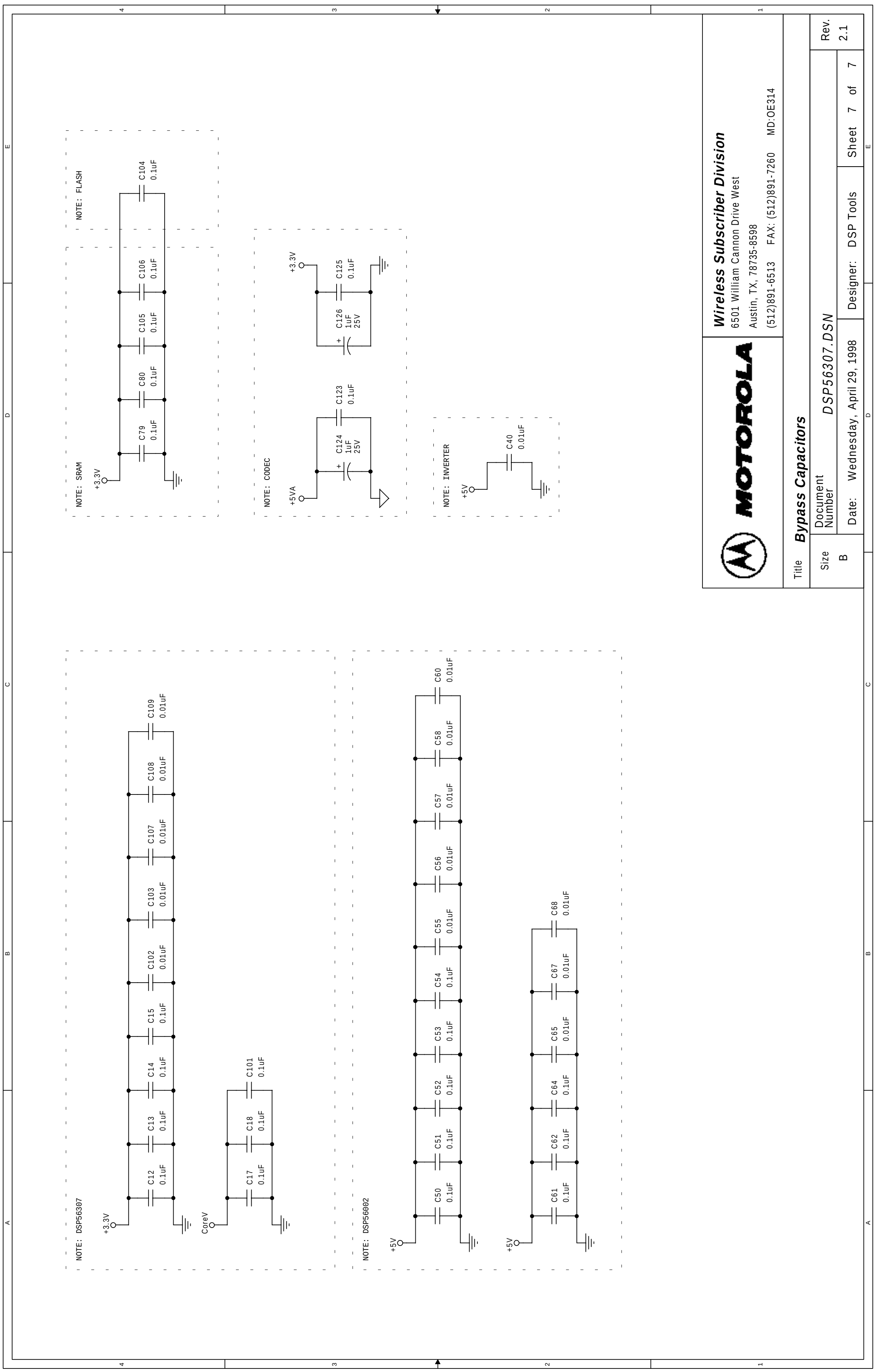


Figure A-7. Bypass Capacitors

Appendix B
DSP56307EVM Parts List

B.1 Parts Listing

The following table contains information on the parts and devices on the DSP56307EVM.

Table B-1. DSP56307EVM Parts List

Designator	Manufacturer	Part Number	Description
U1	Motorola	DSP56307GC100	DSP
U2	Motorola	DSP56002PV80	DSP (JTAG/OnCE)
U4	GSI	GS71024T-10	FSRAM
U3	Atmel	AT29LV010A-20TC	Flash
U5	Quality Semiconductor	QS3384Q	Bus Switch
U6	Motorola	MC34164D-3	Power-On-Reset
U7	Crystal Semiconductor	CS4218-KQ	Audio Codec
U8	Pioneer	LM4880M	Audio Amplifier
U9	Motorola	MC74HCT04AD	Hex Inverter
U11	Maxim	MAX212CAG	RS-232 Transceiver
U23	Motorola	MC33269DT-3.3	3.3 V Regulator
U24	Motorola	MC33269DT-5.0	5 V Regulator
U25	Motorola	MC33269DT	Adj Regulator
D1 D2 D3 D4 D5	Rectron	FM4001	IN4001 Diode
D6 D7	Motorola	MMBD6050LT1	IN6050 Diode
LED1	Quality Technologies	HLMP1790	Green LED
LED2 LED3	Quality Technologies	HLMP1700	Red LED

Table B-1. DSP56307EVM Parts List (Continued)

Designator	Manufacturer	Part Number	Description
Y1	MMD	MC100CA-12.288MHZ	12.288 MHz Oscillator
Y2	ECS	OECS-196.6-3-C3X1A	19.6608 MHz /153.6 kHz Oscillator
SW1 SW2 SW3	Panasonic	EVQ-QS205K	6 mm Switch
P1 P2	Mouser	152-3409	DB-9 Female Connector
P3	Switchcraft	RAPC-722	2.1 mm DC Power Jack
P4 P5 P6	Switchcraft	35RAPC4BHN2	3.5 mm Miniature Stereo Jack
J1	Robinson Nugent	NSH-8DB-S2-TG	Header 8 pin double row
J2	Robinson Nugent	NSH-16DB-S2-TG	Header 16 pin double row
J3	Robinson Nugent	NSH-20DB-S2-TG	Header 20 pin double row
J4 J5	Robinson Nugent	NSH-12DB-S2-TG	Header 12 pin double row
J6	Robinson Nugent	NHS-14DB-S2-TG	Header 14 pin double row
J7 J9	Robinson Nugent	NSH-6SB-S2-TG	Header 6 pin double row
J8	Robinson Nugent	NSH-2SB-S2-TG	Header 2 pin single row
C99 C120 C121	Panasonic	PCS1106CT	10 µF Capacitor, 6.3 V dc
C28 C29 C30 C31 C34 C124 C126	Murata	GRM42-6Y5V105Z025BL	1.0 µF Capacitor, 25 V dc
C9 C12 C13 C14 C15 C17 C18 C35 C37 C50 C51 C52 C53 C54 C61 C62 C64 C79 C80 C96 C101 C104 C105 C106 C110 C122 C123 C125	Murata	GRM40-X7R104K025BL	0.1 µF Capacitor
C16 C40 C55 C56 C57 C58 C60 C65 C67 C68 C102 C103 C107 C108 C109	Murata	GRM40-X7R103K050BL	0.01 µF Capacitor

Table B-1. DSP56307EVM Parts List (Continued)

Designator	Manufacturer	Part Number	Description
C10	Panasonic	PCS1475CT	4.7 µF Capacitor, 6.3 V dc
C2 C3 C4 C23 C24 C39	Murata	GRM42-6Y5V334Z025BL	0.33 µF Capacitor
C21 C22 C25	Murata	GRM42-6Y5V474Z025BL	0.47 µF Capacitor
C38	Murata	GRM42-6Y5V684Z025BL	0.68 µF Capacitor
C19 C20	Xicon	140-CC501N331J	330 pF Capacitor
C26 C27	Murata	GRM40-COG222J050BL	2200 pF Capacitor
C63	Murata	GRM40-X7R821K050BL	820 pF Capacitor
C5 C6 C7 C8	AVX	TPSV476-025R0300	47 µF Capacitor, 10 V dc
C32 C33	Panasonic	PCE3028CT	470 µF Capacitor, 6.3 V dc
C1	Xicon	XAL16V1000	1000 µF Capacitor, 16 V dc
L1 L2 L3 L4 L5	Murata	BL01RN1-A62	**Ferrite Bead**
L6	Murata	LQH4N150K04M00	**Inductor**
R1 R37	NIC	NRC12RF1001TR	1 KΩ Resistor
R3	Xicor	260-5K	5 KΩ Resistor
R4 R6 R7 R8 R9 R11 R12 R13 R14 R15 R18 R19 R20 R21 R22 R31 R32 R33 R34 R35 R36 R38 R39	NIC	NRC12RF1002TR	10 KΩ Resistor
R25 R26 R27 R28	Xicor	260-20K	20 KΩ Resistor
R23 R24	NIC	NRC12RF3922TR	39.2 KΩ Resistor
R16 R17 R29 R30	Xicor	260-5.6K	5.6 KΩ Resistor
R10	NIC	NRC12RF6040TR	604 Ω Resistor
R79 R80	Panasonic	ERJ-6GEYJ240	240 Ω Resistor
R81	NIC	260-4.7K	4.7 KΩ Resistor

Appendix C
Motorola Assembler Notes

C.1 Introduction

This appendix supplements information in Chapter 3 of this document and provides a detailed description of the following components used with the Motorola Assembler:

- Special characters significant to the assembler
- Assembler directives
- Structure control statements

C.2 Assembler Significant Characters

Several one- and two-character sequences are significant to the assembler. The following subsections define these characters and their use.

C.2.1 ; Comment Delimiter Character

Any number of characters preceded by a semicolon (**;**), but not part of a literal string, is considered a comment. Comments are not significant to the assembler, but you can use them to document the source program. Comments are reproduced in the assembler output listing. Comments are normally preserved in macro definitions, but this option can be turned off (see the OPT directive).

Comments can occupy an entire line or can be placed after the last assembler-significant field in a source statement. A comment starting in the first column of the source file is aligned with the label field in the listing file. Otherwise, the comment is shifted right and aligned with the comment field in the listing file.

Example C-1. Example of Comment Delimiter

```
; THIS COMMENT BEGINS IN COLUMN 1 OF THE SOURCE FILE
LOOP      JSR       COMPUTE    ; THIS IS A TRAILING COMMENT
                               ; THESE TWO COMMENTS ARE PRECEDED
                               ; BY A TAB IN THE SOURCE FILE
```

C.2.2 ;; Unreported Comment Delimiter Characters

Unreported comments are any number of characters preceded by two consecutive semicolons (**;;**) that are not part of a literal string. Unreported comments are not considered significant by the assembler and can be included in the source statement, following the same rules as normal comments. However, unreported comments are never reproduced on the assembler output listing and are never saved as part of macro definitions.

Example C-2. Example of Unreported Comment Delimiter

```
;; THESE LINES WILL NOT BE REPRODUCED
;; IN THE SOURCE LISTING
```

C.2.3 \ Line Continuation or Macro Argument Concatenation Character

The following subsections define how the \ character can be used in two different instances.

C.2.3.1 Line Continuation

The backslash character (\), if used as the last character on a line, indicates to the assembler that the source statement continues on the following line. The continuation line is concatenated to the previous line of the source statement, and the result is processed by the assembler as if it were a single-line source statement. The maximum source statement length (the first line and any continuation lines) is 512 characters.

Example C-3. Example of Line Continuation Character

```
; THIS COMMENT \
EXTENDS OVER \
THREE LINES
```

C.2.3.2 Macro Argument Concatenation

The backslash (\) is also used to cause the concatenation of a macro dummy argument with other adjacent alphanumeric characters. For the macro processor to recognize dummy arguments, they must normally be separated from other alphanumeric characters by a non-symbol character. However, sometimes it is desirable to concatenate the argument characters with other characters. If an argument is to be concatenated in front of or behind some other symbol characters, then it must be followed by or preceded by the backslash, respectively.

Example C-4. Example of Macro Concatenation

Suppose the source input file contained the following macro definition:

```
SWAP_REG    MACRO       REG1,REG2  ;swap REG1,REG2 using D4.L as temp
            MOVE        R\REG1,D4.L
            MOVE        R\REG2,R\REG1
            MOVE        D4.L,R\REG2
            ENDM
```

The concatenation operator (\) indicates to the macro processor that the substitution characters for the dummy arguments are to be concatenated in both cases with the character R. If this macro were called with the following statement,

```
            SWAP_REG              0,1
```

the resulting expansion would be

```
            MOVE        R0,D4.L
            MOVE        R1,R0
            MOVE        D4.L,R1
```

C.2.4 ? Return Value of Symbol Character

The **?**<symbol> sequence, when used in macro definitions, is replaced by an ASCII string representing the value of <symbol>. This operator may be used in association with the backslash (\) operator. The value of <symbol> must be an integer (not floating point).

Example C-5. Example of Use of Return Value Character

Consider the following macro definition

```
SWAP_SYM    MACRO       REG1,REG2  ;swap REG1,REG2 using D4.L as temp
            MOVE        R\?REG1,D4.L
            MOVE        R\?REG2,R\?REG1
            MOVE        D4.L,R\?REG2
            ENDM
```

If the source file contained the following SET statements and macro call,

```
AREG        SET         0
BREG        SET         1
            SWAP_SYM    AREG,BREG
```

the resulting expansion as it would appear on the source listing would be

```
            MOVE        R0,D4.L
            MOVE        R1,R0
            MOVE        D4.L,R1
```

C.2.5 % Return Hex Value of Symbol Character

The **%**<symbol> sequence, when used in macro definitions, is replaced by an ASCII string representing the hexadecimal value of <symbol>. This operator may be used in association with the backslash (\) operator. The value of <symbol> must be an integer (not floating point).

Example C-6. Example of Return Hex Value Symbol Character

Consider the following macro definition:

```
GEN_LAB    MACRO      LAB,VAL,STMT
LAB\%VAL   STMT
           ENDM
```

If this macro were called as follows,

```
NUM        SET        10
           GEN_LAB    HEX,NUM,'NOP'
```

The resulting expansion as it would appear in the listing file would be as follows

```
HEXA       NOP
```

C.2.6 ^ Macro Local Label Override

The circumflex (^), when used as a unary expression operator in a macro expansion, causes any local labels in its associated term to be evaluated at normal scope rather than macro scope. This means that any underscore labels in the expression term following the circumflex will not be searched for in the macro local label list. The operator has no effect on normal labels or outside of a macro expansion. The circumflex operator is useful for passing local labels as macro arguments to be used as referents in the macro.

Note: The circumflex is also used as the binary exclusive OR operator.

Example C-7. Example of Local Label Override Character

Consider the following macro definition:

```
LOAD       MACRO      ADDR
           MOVE       P:^ADDR,R0
           ENDM
```

If this macro were called as follows,

```
_LOCAL
           LOAD       _LOCAL
```

the assembler would ordinarily issue an error since _LOCAL is not defined within the body of the macro. With the override operator the assembler recognizes the _LOCAL symbol outside the macro expansion and uses that value in the MOVE instruction.

C.2.7 " Macro String Delimiter or Quoted String DEFINE Expansion Character

The following subsections define how the “ character can be used in two different instances.

C.2.7.1 Macro String

The double quote ("), when used in macro definitions, is transformed by the macro processor into the string delimiter, the single quote (’). The macro processor examines the characters between the double quotes for any macro arguments. This mechanism allows the use of macro arguments as literal strings.

Example C-8. Example of a Macro String Delimiter Character

Using the following macro definition,

```
CSTR        MACRO       STRING
            DC          "STRING"
            ENDM
```

and a macro call,

```
CSTR        ABCD
```

the resulting macro expansion would be

```
            DC          'ABCD'
```

C.2.7.2 Quoted String DEFINE Expansion

A sequence of characters which matches a symbol created with a DEFINE directive is not expanded if the character sequence is contained within a quoted string. Assembler strings generally are enclosed in single quotes (’). If the string is enclosed in double quotes (") then DEFINE symbols are expanded within the string. In all other respects, usage of double quotes is equivalent to that of single quotes.

Example C-9. Example of a Quoted String DEFINE Expression

Consider the source fragment below:

```
           DEFINE     LONG       'short'
STR_MAC    MACRO      STRING
           MSG        'This is a LONG STRING'
           MSG        "This is a LONG STRING"
           ENDM
```

If this macro were invoked as follows,

```
           STR_MAC    sentence
```

then the resulting expansion would be as follows

```
           MSG        'This is a LONG STRING'
           MSG        'This is a short sentence'
```

C.2.8 @ Function Delimiter

All assembler built-in functions start with the (**@**) symbol.

Example C-10. Example of a Function Delimiter Character

```
SVAL       EQ         @SQT(FVAL) ; OBTAIN SQUARE ROOT
```

C.2.9 * Location Counter Substitution

When used as an operand in an expression, the asterisk (*) represents the current integer value of the runtime location counter.

Example C-11. Example of a Location Counter Substitution

```
           ORG        X:$100
XBASE      EQU        *+$20      ; XBASE = $120
```

C.2.10 ++ String Concatenation Operator

Any two strings can be concatenated with the string concatenation operator (++). The two strings must each be enclosed by single or double quotes, and there must be no intervening blanks between the string concatenation operator and the two strings.

Example C-12. Example of a String Concatenation Operator

```
           'ABC'++'DEF' = 'ABCDEF'
```

C.2.11 [ ] Substring Delimiter [<string>,<offset><length>]

Square brackets delimit a substring operation. The <string> argument is the source string. <offset> is the substring starting position within <string>. <length> is the length of the desired substring. <string> may be any legal string combination, including another substring. An error is issued if either <offset> or <length> exceed the length of <string>.

Example C-13. Example of a Substring Delimiter

```
            DEFINE     ID         ['DSP56000',3,5]; ID = '56000'
```

C.2.12 << I/O Short Addressing Mode Force Operator

Many DSP instructions allow an I/O short form of addressing. If the value of an absolute address is known to the assembler on pass one, then the assembler will always pick the shortest form of addressing consistent with the instruction format. If the absolute address is not known to the assembler on pass one (that is, the address is a forward or external reference), then the assembler picks the long form of addressing by default. If this is not desired, then the I/O short form of addressing can be forced by preceding the absolute address by the I/O short addressing mode force operator (<<).

Example C-14. Example of an I/O Short Addressing Mode Force Operator

Since the symbol IOPORT is a forward reference in the following sequence of source lines, the assembler would pick the long absolute form of addressing by default:

```
            BTST       #4,Y:IOPORT
IOPORT      EQU        Y:$FFF3
```

Because the long absolute addressing mode would cause the instruction to be two words long instead of one word for the I/O short absolute addressing mode, it would be desirable to force the I/O short absolute addressing mode as shown below:

```
            BTST       #4,Y:<<IOPORT
IOPORT      EQU        Y:$FFF3
```

C.2.13 < Short Addressing Mode Force Operator

Many DSP instructions allow a short form of addressing. If the value of an absolute address is known to the assembler on pass one, or the FORCE SHORT directive is active, then the assembler will always pick the shortest form of addressing consistent with the instruction format. If the absolute address is not known to the assembler on pass one (that is, the address is a forward or external reference), then the assembler picks the long form of addressing by default. If this is not desired, then the short absolute form of addressing

can be forced by preceding the absolute address by the short addressing mode force operator (<).

Example C-15. Example of a Short Addressing Mode Force Operator

Since the symbol DATAST is a forward reference in the following sequence of source lines, the assembler would pick the long absolute form of addressing by default:

```
            MOVE        D0.L,Y:DATAST
DATAST      EQU         Y:$23
```

Because the long absolute addressing mode would cause the instruction to be two words long instead of one word for the short absolute addressing mode, it would be desirable to force the short absolute addressing mode as shown below:

```
            MOVE        D0.L,:<DATAST
DATAST      EQU         Y:$23
```

C.2.14 > Long Addressing Mode Force Operator

Many DSP instructions allow a long form of addressing. If the value of an absolute address is known to the assembler on pass one, then the assembler will always pick the shortest form of addressing consistent with the instruction format, unless the FORCE LONG directive is active. If this is not desired, then the long absolute form of addressing can be forced by preceding the absolute address by the long addressing mode force operator (>).

Example C-16. Example of a Long Addressing Mode Force Operator

Since the symbol DATAST is a not a forward reference in the following sequence of source lines, the assembler would pick the short absolute form of addressing:

```
DATAST      EQU         Y:$23
            MOVE        D0.L,Y:DATAST
```

If this is not desirable, then the long absolute addressing mode can be forced as shown below:

```
DATAST      EQU         Y:$23
            MOVE        D0.L,Y:>DATAST
```

C.2.15 # Immediate Addressing Mode

The pound sign (#) is used to indicate to the assembler to use the immediate addressing mode.

Example C-17. Example of Immediate Addressing Mode

```
CNST        EQU         $5
            MOVE        #CNST,D0.L
```

C.2.16 #< Immediate Short Addressing Mode Force Operator

Many DSP instructions allow a short immediate form of addressing. If the immediate data is known to the assembler on pass one (not a forward or external reference) or the FORCE SHORT directive is active, then the assembler will always pick the shortest form of immediate addressing consistent with the instruction. If the immediate data is a forward or external reference, then the assembler picks the long form of immediate addressing by default. If this is not desired, then the short form of addressing can be forced using the immediate short addressing mode force operator (#<).

Example C-18. Example of Immediate Short Addressing Mode Force Operator

In the following sequence of source lines, the symbol CNST is not known to the assembler on pass one, and therefore, the assembler would use the long immediate addressing form for the MOVE instruction.

```
            MOVE        #CNST,D0.L
CNST        EQU         $5
```

Because the long immediate addressing mode makes the instruction two words long instead of one word for the immediate short addressing mode, it may be desirable to force the immediate short addressing mode as shown below:

```
            MOVE        #<CNST,D0.L
CNST        EQU         $5
```

C.2.17 #> Immediate Long Addressing Mode Force Operator

Many DSP instructions allow a long immediate form of addressing. If the immediate data is known to the assembler on pass one (not a forward or external reference), then the assembler will always pick the shortest form of immediate addressing consistent with the instruction, unless the FORCE LONG directive is active. If this is not desired, then the long form of addressing can be forced using the immediate long addressing mode force operator (#>).

Example C-19. Example of an Immediate Long Addressing Mode Operator

In the following sequence of source lines, the symbol CNST is known to the assembler on pass one, and therefore, the assembler would use the short immediate addressing form for the MOVE instruction.

```
CNST        EQU         $5
            MOVE        #CNST,D0.L
```

If this is not desirable, then the long immediate form of addressing can be forced as shown below:

```
CNST        EQU         $5
            MOVE        #>CNST,D0.L
```

C.3 Assembler Directives

The following subsections define each directive and its use.

C.3.1 BADDR Set Buffer Address

```
            BADDR       <M | R>,<expression>
```

The BADDR directive sets the runtime location counter to the address of a buffer of the given type, the length of which in words is equal to the value of <expression>. The buffer type may be either modulo or reverse-carry. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where 2^k >= <expression>. An error is issued if there is insufficient memory remaining to establish a valid base address. Unlike other buffer allocation directives, the runtime location counter is not advanced by the value of the integer expression in the operand field; the location counter remains at the buffer base address. The block of memory intended for the buffer is not initialized to any value.

The result of <expression> may have any memory space attribute but must be an absolute integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). If a modulo buffer is specified, the expression must fall within the range 2 ≤ <expression> ≤ *m*, where *m* is the maximum address of the target DSP. If a reverse-carry buffer is designated and <expression> is not a power of two, a warning is issued. A label is not allowed with this directive.

Note: See also BSM, BSB, BUFFER, DSM, DSR.

Example C-20. Example BADDR Directive

```
            ORG         X:$100
M_BUF       BADDR       M,24        ; CIRCULAR BUFFER MOD 24
```

C.3.2 BSB Block Storage Bit-Reverse

```
[<label>]   BSB      <expression>[,<expression>]
```

The BSB directive causes the assembler to allocate and initialize a block of words for a reverse-carry buffer. The number of words in the block is given by the first expression, which must evaluate to an absolute integer. Each word is assigned the initial value of the second expression. If there is no second expression, an initial value of zero is assumed. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where 2^k is greater than or equal to the value of the first expression. An error will occur if the first expression contains symbols that are not yet defined (forward references) or if the expression has a value of less than or equal to zero. Also, if the first expression is not a power of two a warning is generated. Both expressions can have any memory space attribute.

<label>, if present, is assigned the value of the runtime location counter after a valid base address has been established.

Only one word of object code is shown on the listing, regardless of how large the first expression is. However, the runtime location counter is advanced by the number of words generated.

Note: See also BSC, BSM, DC.

Example C-21. Buffer Directive

```
BUFFER      BSB        BUFSIZ    ; INITIALIZE BUFFER TO ZEROS
```

C.3.3 BSC Block Storage of Constant

```
[<label>]   BSC      <expression>[,<expression>]
```

The BSC directive causes the assembler to allocate and initialize a block of words. The number of words in the block is given by the first expression, which must evaluate to an absolute integer. Each word is assigned the initial value of the second expression. If there is no second expression, an initial value of zero is assumed. If the first expression contains symbols that are not yet defined (forward references) or if the expression has a value of less than or equal to zero, an error is generated. Both expressions can have any memory space attribute.

<label>, if present, is assigned the value of the runtime location counter at the start of the directive processing.

Only one word of object code is shown on the listing, regardless of how large the first expression is. However, the runtime location counter is advanced by the number of words generated.

Note: See also BSM, BSB, DC.

Example C-22. Block Storage of Constant Directive

```
UNUSED      BSC       $2FFF-@LCV(R),$FFFFFFFF; FILL UNUSED EPROM
```

C.3.4 BSM Block Storage Modulo

```
[<label>]   BSM      <expression>[,<expression>]
```

The BSM directive causes the assembler to allocate and initialize a block of words for a modulo buffer. The number of words in the block is given by the first expression, which must evaluate to an absolute integer. Each word is assigned the initial value of the second expression. If there is no second expression, an initial value of zero is assumed. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where 2^k is greater than or equal to the value of the first expression. An error will occur if the first expression contains symbols that are not yet defined (forward references), has a value of less than or equal to zero, or falls outside the range 2 ≤ <expression> ≤ *m*, where *m* is the maximum address of the target DSP. Both expressions may have any memory space attribute.

<label>, if present, is assigned the value of the runtime location counter after a valid base address has been established.

Only one word of object code is shown on the listing, regardless of how large the first expression is. However, the runtime location counter is advanced by the number of words generated.

Note: See also BSC, BSB, DC.

Example C-23. Block Storage Modulo Directive

```
BUFFER      BSM       BUFSIZ,$FFFFFFFF; INITIALIZE BUFFER TO ALL ONES
```

C.3.5 BUFFER Start Buffer

```
BUFFER    <M | R>,<expression>
```

The BUFFER directive indicates the start of a buffer of the given type. Data is allocated for the buffer until an ENDBUF directive is encountered. Instructions and most data definition directives may appear between the BUFFER and ENDBUF pair, although BUFFER directives may not be nested and certain types of directives such as MODE, ORG, SECTION, and other buffer allocation directives may not be used. The <expression> represents the buffer size. If less data is allocated than the size of the buffer, the remaining buffer locations are uninitialized. If more data is allocated than the specified size of the buffer, an error is issued.

The BUFFER directive sets the runtime location counter to the address of a buffer of the given type, the length of which in words is equal to the value of <expression>. The buffer type may be either modulo or reverse-carry. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where 2^k >= <expression>. An error is issued if there is insufficient memory remaining to establish a valid base address. Unlike other buffer allocation directives, the runtime location counter is not advanced by the value of the integer expression in the operand field; the location counter remains at the buffer base address.

The result of <expression> may have any memory space attribute but must be an absolute integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). If a modulo buffer is specified, the expression must fall within the range 2 ≤ <expression> ≤ *m*, where *m* is the maximum address of the target DSP. If a Reverse-carry buffer is designated and <expression> is not a power of two a warning is issued.

Note: A label is not allowed with this directive. See also BADDR, BSM, BSB, DSM, DSR, ENDBUF.

Example C-24. Buffer Directive

```
           ORG        X:$100
           BUFFER     M,24        ; CIRCULAR BUFFER MOD 24
M_BUF      DC         0.5,0.5,0.5,0.5
           DS         20          ; REMAINDER UNINITIALIZED
           ENDBUF
```

C.3.6 COBJ Comment Object File

```
           COBJ       <string>
```

The COBJ directive is used to place a comment in the object code file. The <string> is put in the object file as a comment.

Note: A label is not allowed with this directive. See also IDENT.

Example C-25. COBM Directive

```
           COBJ       'Start of filter coefficients'
```

C.3.7 COMMENT Start Comment Lines

```
COMMENT    <delimiter>
.
.
<delimiter>
```

The COMMENT directive is used to define one or more lines as comments. The first non-blank character after the COMMENT directive is the comment delimiter. The two delimiters are used to define the comment text. The line containing the second comment delimiter is considered the last line of the comment. The comment text can include any printable characters and the comment text is reproduced in the source listing as it appears in the source file.

Note: A label is not allowed with this directive.

Example C-26. COMMENT Directive

```
COMMENT    +      This is a one line comment +
COMMENT    *      This is a multiple line comment. Any
                  number of lines can be placed between the
                  two delimiters.
           *
```

C.3.8 DC Define Constant

```
<label>]   DC        <arg>[,<arg>,...,<arg>]
```

The DC directive allocates and initializes a word of memory for each <arg> argument. <arg> may be a numeric constant, a single or multiple character string constant, a symbol, or an expression. The DC directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location is filled with zeros. If the DC directive is used in L memory, the arguments are evaluated and stored as long word quantities. Otherwise, an error occurs if the evaluated argument value is too large to represent in a single DSP word.

<label>, if present, is assigned the value of the runtime location counter at the start of the directive processing.

Integer arguments are stored as is; floating point numbers are converted to binary values. Single and multiple character strings are handled in the following manner:

- Single character strings are stored in a word whose lower seven bits represent the ASCII value of the character.

Example C-27. Single Character String Definition

```
          'R'          = $000052
```

- Multiple character strings represent words whose bytes are composed of concatenated sequences of the ASCII representation of the characters in the string (unless the NOPS option is specified; see the OPT directive). If the number of characters is not an even multiple of the number of bytes per DSP word, then the last word will have the remaining characters left aligned and the rest of the word is zero-filled. If the NOPS option is given, each character in the string is stored in a word whose lower seven bits represent the ASCII value of the character.

Example C-28. Multiple Character String Definition

```
'ABCD'       =          $414243
                        $440000
```

Note: See also BSC, DCB.

Example C-29. DC Directive

```
TABLE      DC          1426,253,$2662,'ABCD'
CHARS      DC          'A','B','C','D'
```

C.3.9 DCB Define Constant Byte

```
[<label>]   DCB        <arg>[,<arg>,...,<arg>]
```

The DCB directive allocates and initializes a byte of memory for each <arg> argument. <arg> may be a byte integer constant, a single or multiple character string constant, a symbol, or a byte expression. The DCB directive may have one or more arguments separated by commas. Multiple arguments are stored in successive byte locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding byte location is filled with zeros.

<label>, if present, is assigned the value of the runtime location counter at the start of the directive processing.

Integer arguments are stored as is but must be byte values (i.e., within the range 0–255); floating point numbers are not allowed. Single and multiple character strings are handled in the following manner:

- Single character strings are stored in a word whose lower seven bits represent the ASCII value of the character. (See **Example C-27.**.)
- Multiple character strings represent words whose bytes are composed of concatenated sequences of the ASCII representation of the characters in the string (unless the NOPS option is specified; see the OPT directive). If the number of characters is not an even multiple of the number of bytes per DSP word, then the last word will have the remaining characters left aligned and the rest of the word

is zero-filled. If the NOPS option is given, each character in the string is stored in a word whose lower seven bits represent the ASCII value of the character. (See **Example C-28.**.)

Note: See also BSC, DC.

Example C-30. DCB Directive

```
TABLE       DCB         'two',0,'strings',0
CHARS       DCB         'A','B','C','D'
```

C.3.10 DEFINE Define Substitution String

```
            DEFINE      <symbol>   <string>
```

The DEFINE directive is used to define substitution strings that are used on all following source lines. All succeeding lines are searched for an occurrence of <symbol>, which is replaced by <string>. This directive is useful for providing better documentation in the source program. <symbol> must adhere to the restrictions for non-local labels. That is, it cannot exceed 512 characters, the first of which must be alphabetic, and the remainder of which must be either alphanumeric or the underscore(_). A warning results if a new definition of a previously defined symbol is attempted. The assembler output listing will show lines after the DEFINE directive has been applied and therefore redefined symbols are replaced by their substitution strings unless the NODXL option is in effect. (See **C.3.42 OPT Assembler Options** on page -34.)

Macros represent a special case. DEFINE directive translations are applied to the macro definition as it is encountered. When the macro is expanded any active DEFINE directive translations are applied again.

DEFINE directive symbols that are defined within a section will only apply to that section. See the SECTION directive.

Note: A label is not allowed with this directive. See also UNDEF**.**

Example C-31. DEFINE Directive

If the following DEFINE directive occurred in the first part of the source program:

```
            DEFINE      ARRAYSIZ   '10 * SAMPLSIZ'
```

then the source line below:

```
            DS          ARRAYSIZ
```

would be transformed by the assembler to the following:

```
            DS          10 * SAMPLSIZ
```

C.3.11 DS Define Storage

```
[<label>]   DS        <expression>
```

The DS directive reserves a block of memory the length of which in words is equal to the value of <expression>. This directive causes the runtime location counter to be advanced by the value of the absolute integer expression in the operand field. <expression> can have any memory space attribute. The block of memory reserved is not initialized to any value. The expression must be an integer greater than zero and cannot contain any forward references (symbols that have not yet been defined).

<label>, if present, is assigned the value of the runtime location counter at the start of the directive processing.

Note: See also DSM, DSR.

Example C-32. DS Directive

```
S_BUF      DS         12         ; SAMPLE BUFFER
```

C.3.12 DSM Define Modulo Storage

```
[<label>]   DSM       <expression>
```

The DSM directive reserves a block of memory the length of which in words is equal to the value of <expression>. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where 2^k >= <expression>. An error is issued if there is insufficient memory remaining to establish a valid base address. Next the runtime location counter is advanced by the value of the integer expression in the operand field. <expression> can have any memory space attribute. The block of memory reserved is not initialized to any given value. The result of <expression> must be an absolute integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). The expression also must fall within the range:

$2 \leq$ <expression> $\leq m$,
where m is the maximum address of the target DSP.

<label>, if present, is assigned the value of the runtime location counter after a valid base address has been established.

Note: See also DS, DSR.

Example C-33. DSM Directive

```
           ORG         X:$100
M_BUF      DSM         24         ; CIRCULAR BUFFER MOD 24
```

C.3.13 DSR Define Reverse Carry Storage

```
[<label>]   DSR       <expression>
```

The DSR directive reserves a block of memory the length of which in words is equal to the value of <expression>. If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where $2^k \geq$ <expression>. An error is issued if there is insufficient memory remaining to establish a valid base address. Next the runtime location counter is advanced by the value of the integer expression in the operand field. <expression> can have any memory space attribute. The block of memory reserved is not initialized to any given value. The result of <expression> must be an absolute integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). Because the DSR directive is useful mainly for generating FFT buffers, if <expression> is not a power of two a warning is generated.

<label>, if present, is assigned the value of the runtime location counter after a valid base address has been established.

Note: See also DS, DSM.

Example C-34. DSR Directive

```
        ORG     X:$100
R_BUF   DSR     8          ; REVERSE CARRY BUFFER FOR 16 POINT FFT
```

C.3.14 DUP Duplicate Sequence of Source Lines

```
[<label>]   DUP       <expression>
           .
           .
            ENDM
```

The sequence of source lines between the DUP and ENDM directives are duplicated by the number specified by the integer <expression>. <expression> can have any memory space attribute. If the expression evaluates to a number less than or equal to zero, the sequence of lines will not be included in the assembler output. The expression result must be an absolute integer and cannot contain any forward references (symbols that have not already been defined). The DUP directive may be nested to any level.

<label>, if present, is assigned the value of the runtime location counter at the start of the DUP directive processing.

Note: See also DUPA, DUPC, DUPF, ENDM, MACRO.

Example C-35. DUP Directive

The sequence of source input statements,

```
COUNT      SET      3
           DUP      COUNT      ; ASR BY COUNT
           ASR      D0
           ENDM
```

would generate the following in the source listing:

```
COUNT      SET      3
           DUP      COUNT      ; ASR BY COUNT
           ASR      D0
           ASR      D0
           ASR      D0
           ENDM
```

Note that the lines

```
           DUP      COUNT      ;ASR BY COUNT
           ENDM
```

will only be shown on the source listing if the MD option is enabled. The lines

```
           ASR      D0
           ASR      D0
           ASR      D0
```

will only be shown on the source listing if the MEX option is enabled.

Note: See the OPT directive in this appendix for more information on the MD and MEX options.

C.3.15 DUPA Duplicate Sequence With Arguments

```
           [<label>]  DUPA      <dummy>,<arg>[<,<arg>,...,<arg>]
                      .
                      .
                      ENDM
```

The block of source statements defined by the DUPA and ENDM directives are repeated for each argument. For each repetition, every occurrence of the dummy parameter within the block is replaced with each succeeding argument string. If the argument string is a null, then the block is repeated with each occurrence of the dummy parameter removed. If an argument includes an embedded blank or other assembler-significant character, it must be enclosed with single quotes.

<label>, if present, is assigned the value of the runtime location counter at the start of the DUPA directive processing.

Note: See also DUP, DUPC, DUPF, ENDM, MACRO.

Example C-36. DUPA Directive

If the input source file contained the following statements,

```
DUPA         VALUE,12,32,34
DC           VALUE
ENDM
```

then the assembled source listing would show

```
DUPA         VALUE,12,32,34
DC           12
DC           32
DC           34
ENDM
```

Note that the lines

```
DUPA         VALUE,12,32,34
ENDM
```

will only be shown on the source listing if the MD option is enabled. The lines

```
DC           12
DC           32
DC           34
```

will only be shown on the source listing if the MEX option is enabled.

Note: See the OPT directive in this appendix for more information on the MD and MEX options.

C.3.16 DUPC Duplicate Sequence With Characters

```
[<label>]    DUPC       <dummy>,<string>
            .
            .
             ENDM
```

The block of source statements defined by the DUPC and ENDM directives are repeated for each character of <string>. For each repetition, every occurrence of the dummy parameter within the block is replaced with each succeeding character in the string. If the string is null, then the block is skipped.

<label>, if present, is assigned the value of the runtime location counter at the start of the DUPC directive processing.

Note: See also DUP, DUPA, DUPF, ENDM, MACRO.

Example C-37. DUPC Directive

If input source file contained the following statements,

```
DUPC        VALUE,'123'
DC          VALUE
ENDM
```

then the assembled source listing would show:

```
DUPC        VALUE,'123'
DC          1
DC          2
DC          3
ENDM
```

Note that the lines

```
DUPC        VALUE,'123'
ENDM
```

will only be shown on the source listing if the MD option is enabled. The lines

```
DC          1
DC          2
DC          3
```

will only be shown on the source listing if the MEX option is enabled.

Note: See the OPT directive in this appendix for more information on the MD and MEX options.

C.3.17 DUPF Duplicate Sequence In Loop

```
[<label>]   DUPF      <dummy>,[<start>],<end>[,<increment>]
           .
           .
            ENDM
```

The block of source statements defined by the DUPF and ENDM directives are repeated in general (<end> – <start>) + 1 times when <increment> is 1. <start> is the starting value for the loop index; <end> represents the final value. <increment> is the increment for the loop index; it defaults to 1 if omitted (as does the <start> value). The <dummy> parameter holds the loop index value and may be used within the body of instructions.

<label>, if present, is assigned the value of the runtime location counter at the start of the DUPF directive processing.

Note: See also DUP, DUPA, DUPC, ENDM, MACRO.

Example C-38. DUPF Directive

If input source file contained the following statements,

```
DUPF      NUM,0,7
MOVE      #0,R\NUM
ENDM
```

then the assembled source listing shows:

```
DUPF      NUM,0,7
MOVE      #0,R0
MOVE      #0,R1
MOVE      #0,R2
MOVE      #0,R3
MOVE      #0,R4
MOVE      #0,R5
MOVE      #0,R6
MOVE      #0,R7
ENDM
```

Note that the lines

```
DUPF      NUM,0,7
ENDM
```

are only shown on the source listing if the MD option is enabled. The lines

```
MOVE      #0,R0
MOVE      #0,R1
MOVE      #0,R2
MOVE      #0,R3
MOVE      #0,R4
MOVE      #0,R5
MOVE      #0,R6
MOVE      #0,R7
```

are only shown on the source listing if the MEX option is enabled.

Note: See the OPT directive in this appendix for more information on the MD and MEX options.

C.3.18 END End of Source Program

```
END       [<expression>]
```

The optional END directive indicates that the logical end of the source program has been encountered. Any statements following the END directive are ignored. The optional expression in the operand field can be used to specify the starting execution address of the

program. <expression> may be absolute or relocatable but must have a memory space attribute of Program or None. The END directive cannot be used in a macro expansion.

Note: A label is not allowed with this directive.

Example C-39. END Directive

```
              END        BEGIN   ; BEGIN is the starting execution address
```

C.3.19 ENDBUF End Buffer

```
              ENDBUF
```

The ENDBUF directive is used to signify the end of a buffer block. The runtime location counter will remain just beyond the end of the buffer when the ENDBUF directive is encountered.

Note: A label is not allowed with this directive. See also BUFFER**.**

Example C-40. ENDBUF Directive

```
              ORG        X:$100
BUF           BUFFER     R,64        ; uninitialized reverse-carry buffer
              ENDBUF
```

C.3.20 ENDIF End of Conditional Assembly

```
              ENDIF
```

The ENDIF directive is used to signify the end of the current level of conditional assembly. Conditional assembly directives can be nested to any level, but the ENDIF directive always refers to the most previous IF directive.

Note: A label is not allowed with this directive. (See **C.3.31 IF Conditional Assembly Directive** on page -28.)**.**

Example C-41. ENDIF Directive

```
              IF         @REL()
SAVEPC        SET        *           ; Save current program counter
              ENDIF
```

C.3.21 ENDM End of Macro Definition

```
              ENDM
```

Every MACRO, DUP, DUPA, and DUPC directive must be terminated by an ENDM directive.

Note: A label is not allowed with this directive. See also DUP, DUPA, DUPC, MACRO.

Example C-42. ENDM Directive

```
SWAP_SYM    MACRO       REG1,REG   ;swap REG1,REG2 using D4.L as temp
            MOVE        R\?REG1,D4.L
            MOVE        R\?REG2,R\?REG1
            MOVE        D4.L,R\?REG2
            ENDM
```

C.3.22 ENDSEC End Section

```
            ENDSEC
```

Every SECTION directive must be terminated by an ENDSEC directive.

Note: A label is not allowed with this directive. See also SECTION.

Example C-43. ENDSEC Directive

```
            SECTION     COEFF
            ORG         Y:
VALUES      BSC         $100       ; Initialize to zero
            ENDSEC
```

C.3.23 EQU Equate Symbol to a Value

```
            <label>     EQU [{X: | Y: | L: | P: | E:}]<expression>
```

The EQU directive assigns the value and memory space attribute of <expression> to the symbol <label>. If <expression> has a memory space attribute of None, then it can optionally be preceded by any of the indicated memory space qualifiers to force a memory space attribute. An error will occur if the expression has a memory space attribute other than None and it is different than the forcing memory space attribute. The optional forcing memory space attribute is useful to assign a memory space attribute to an expression that consists only of constants but is intended to refer to a fixed address in a memory space.

The EQU directive is one of the directives that assigns a value other than the program counter to the label. The label cannot be redefined anywhere else in the program (or section, if SECTION directives are being used). The <expression> may be relative or absolute but cannot include a symbol that is not yet defined (no forward references are allowed).

Note: See also SET.

Example C-44. EQU Directive

```
A_D_PORT    EQU         X:$4000
```

This assigns the value $4000 with a memory space attribute of **X** to the symbol A_D_PORT.

```
COMPUTE     EQU         @LCV(L)
```

@LCV(L) is used to refer to the value and memory space attribute of the load location counter. This value and memory space attribute is assigned to the symbol COMPUTE.

C.3.24 EXITM Exit Macro

```
            EXITM
```

The EXITM directive will cause immediate termination of a macro expansion. It is useful when used with the conditional assembly directive IF to terminate macro expansion when error conditions are detected.

Note: A label is not allowed with this directive. See also DUP, DUPA, DUPC, MACRO**.**

Example C-45. EXITM Directive

```
CALC        MACRO       XVAL,YVAL
            IF          XVAL<0
            FAIL        'Macro parameter value out of range'
            EXITM       ; Exit macro
            ENDIF
            .
            .
            .
            ENDM
```

C.3.25 FAIL Programmer Generated Error

```
     FAIL      [{<str>|<exp>}[,{<str>|<exp>},...,{<str>|<exp>}]]
```

The FAIL directive will cause an error message to be output by the assembler. The total error count is incremented as with any other error. The FAIL directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the error has been printed. An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified optionally to describe the nature of the generated error.

Note: A label is not allowed with this directive. See also MSG, WARN**.**

Example C-46. FAIL Directive

```
            FAIL        'Parameter out of range'
```

C.3.26 FORCE Set Operand Forcing Mode

```
            FORCE       {SHORT | LONG | NONE}
```

The FORCE directive causes the assembler to force all immediate, memory, and address operands to the specified mode as if an explicit forcing operator were used. Note that if a relocatable operand value forced short is determined to be too large for the instruction word, an error will occur at link time, not during assembly. Explicit forcing operators override the effect of this directive.

Note: A label is not allowed with this directive. See also **<**, **>**, **#<**, **#>.**

Example C-47. FORCE Directive

```
            FORCE       SHORT       ; force operands short
```

C.3.27 GLOBAL Global Section Symbol Declaration

```
            GLOBAL      <symbol>[,<symbol>,...,<symbol>]
```

The GLOBAL directive is used to specify that the list of symbols is defined within the current section, and that those definitions should be accessible by all sections. This directive is only valid if used within a program block bounded by the SECTION and ENDSEC directives. If the symbols that appear in the operand field are not defined in the section, an error is generated.

Note: A label is not allowed with this directive. See also SECTION, XDEF, XREF**.**

Example C-48. GLOBAL Directive

```
   SECTION IO
   GLOBAL  LOOPA  ; LOOPA will be globally accessible by other sections
   .
   .
   .
   ENDSEC
```

C.3.28 GSET Set Global Symbol to a Value

```
            <label>     GSET        <expression>
                        GSET        <label>     <expression>
```

The GSET directive is used to assign the value of the expression in the operand field to the label. The GSET directive functions somewhat like the EQU directive. However, labels defined via the GSET directive can have their values redefined in another part of the program (but only through the use of another GSET or SET directive). The GSET

directive is useful for resetting a global SET symbol within a section, where the SET symbol would otherwise be considered local. The expression in the operand field of a GSET must be absolute and cannot include a symbol that is not yet defined (no forward references are allowed).

Note: See also EQU, SET.

Example C-49. GSET Directive

```
COUNT      GSET       0          ; INITIALIZE COUNT
```

C.3.29 HIMEM Set High Memory Bounds

```
           HIMEM      <mem>[<rl>]:<expression>[,...]
```

The HIMEM directive establishes an absolute high memory bound for code and data generation. <mem> corresponds to one of the DSP memory spaces (X, Y, L, P, E). <rl> is one of the letters R for runtime counter or L for load counter. The <expression> is an absolute integer value within the address range of the machine. If during assembly the specified location counter exceeds the value given by <expression>, a warning is issued.

Note: A label is not allowed with this directive. See also LOMEM**.**

Example C-50. HIMEM Directive

```
HIMEM                 XR:$7FFF,YR:$7FFF    ; SET X/Y RUN HIGH MEM
BOUNDS
```

C.3.30 IDENT Object Code Identification Record

```
           [<label>]  IDENT      <expression1>,<expression2>
```

The IDENT directive is used to create an identification record for the object module. If <label> is specified, it is used as the module name. If <label> is not specified, then the filename of the source input file is used as the module name. <expression1> is the version number; <expression2> is the revision number. The two expressions must each evaluate to an integer result. The comment field of the IDENT directive is also passed on to the object module.

Note: See also COBJ**.**

Example C-51. IDENT Directive

If the following line was included in the source file,

```
FFILTER    IDENT     1,2       ; FIR FILTER MODULE
```

then the object module identification record includes the module name (FFILTER), the version number (1), the revision number (2), and the comment field (; FIR FILTER MODULE).

C.3.31 IF Conditional Assembly Directive

```
IF        <expression>
.
.
[ELSE]    (the ELSE directive is optional)
.
.
ENDIF
```

Part of a program that is to be conditionally assembled must be bounded by an IF-ENDIF directive pair. If the optional ELSE directive is not present, then the source statements following the IF directive and up to the next ENDIF directive is included as part of the source file being assembled only if the <expression> has a nonzero result. If the <expression> has a value of zero, the source file is assembled as if those statements between the IF and the ENDIF directives were never encountered. If the ELSE directive is present and <expression> has a nonzero result, then the statements between the IF and ELSE directives are assembled, and the statements between the ELSE and ENDIF directives are skipped. Alternatively, if <expression> has a value of zero, then the statements between the IF and ELSE directives are skipped, and the statements between the ELSE and ENDIF directives are assembled.

The <expression> must have an absolute integer result and is considered true if it has a nonzero result. The <expression> is false only if it has a result of zero. Because of the nature of the directive, <expression> must be known on pass one (no forward references allowed). IF directives can be nested to any level. The ELSE directive will always refer to the nearest previous IF directive as will the ENDIF directive.

Note: A label is not allowed with this directive. See also ENDIF.

Example C-52. IF Directive

```
        IF          @LST>0
        DUP         @LST        ; Unwind LIST directive stack
        NOLIST
        ENDM
        ENDIF
```

C.3.32 INCLUDE Include Secondary File

```
INCLUDE    <string>  |  <<string>>
```

This directive is inserted into the source program at any point where a secondary file is to be included in the source input stream. The string specifies the filename of the secondary file. The filename must be compatible with the operating system and can include a directory specification. If no extension is given for the filename, a default extension of .ASM is supplied.

The file is searched for first in the current directory, unless the <<string>> syntax is used, or in the directory specified in <string>. If the file is not found, and the -I option was used on the command line that invoked the assembler, then the string specified with the -I option is prefixed to <string> and that directory is searched. If the <<string>> syntax is given, the file is searched for only in the directories specified with the -I option.

Note: A label is not allowed with this directive. See also MACLIB**.**

Example C-53. INCLUDE Directive

```
        INCLUDE  'headers/io.asm'; Unix example
        INCLUDE  'storage\mem.asm'; MS-DOS example
        INCLUDE <data.asm>    ; Do not look in current directory
```

C.3.33 LIST List the Assembly

```
LIST
```

Print the listing from this point on. The LIST directive is not printed, but the subsequent source lines are output to the source listing. The default is to print the source listing. If the IL option has been specified, the LIST directive has no effect when encountered within the source program.

The LIST directive actually increments a counter that is checked for a positive value and is symmetrical with respect to the NOLIST directive.

Note the following sequence:

```
; Counter value currently 1
LIST                  ; Counter value = 2
LIST                  ; Counter value = 3
NOLIST                ; Counter value = 2
NOLIST                ; Counter value = 1
```

The listing still would not be disabled until another NOLIST directive was issued.

Note: A label is not allowed with this directive. See also NOLIST, OPT.

Example C-54. LIST Directive

```
IF         LISTON
LIST                  ; Turn the listing back on
ENDIF
```

C.3.34 LOCAL Local Section Symbol Declaration

```
LOCAL        <symbol>[,<symbol>,...,<symbol>]
```

The LOCAL directive is used to specify that the list of symbols is defined within the current section, and that those definitions are explicitly local to that section. It is useful in cases where a symbol is used as a forward reference in a nested section where the enclosing section contains a like-named symbol. This directive is only valid if used within a program block bounded by the SECTION and ENDSEC directives. The LOCAL directive must appear before <symbol> is defined in the section. If the symbols that appear in the operand field are not defined in the section, an error is generated.

Note: A label is not allowed with this directive. See also SECTION, XDEF, XREF.

Example C-55. LOCAL Directives

```
SECTION    IO
LOCAL      LOOPA      ; LOOPA local to this section
.
.
.
ENDSEC
```

C.3.35 LOMEM Set Low Memory Bounds

```
LOMEM      <mem>[<rl>]:<expression>[,...]
```

The LOMEM directive establishes an absolute low memory bound for code and data generation. <mem> corresponds to one of the DSP memory spaces (X, Y, L, P, E). <rl> is one of the letters R for runtime counter or L for load counter. The <expression> is an absolute integer value within the address range of the machine. If during assembly the

specified location counter falls below the value given by <expression>, a warning is issued.

Note: A label is not allowed with this directive. See also HIMEM**.**

Example C-56. LOMEM Directive

```
LOMEM                      XR:$100,YR:$100; SET X/Y RUN LOW MEM BOUNDS
```

C.3.36 LSTCOL Set Listing Field Widths

```
LSTCOL  [<labw>[,<opcw>[,<oprw>[,<opc2w>[,<opr2w>[,<xw>[,<yw>]]]]]]]
```

Sets the width of the output fields in the source listing. Widths are specified in terms of column positions. The starting position of any field is relative to its predecessor except for the label field, which always starts at the same position relative to page left margin, program counter value, and cycle count display. The widths may be expressed as any positive absolute integer expression. However, if the width is not adequate to accommodate the contents of a field, the text is separated from the next field by at least one space.

Any field for which the default is desired may be null. A null field can be indicated by two adjacent commas with no intervening space or by omitting any trailing fields altogether. If the LSTCOL directive is given with no arguments all field widths are reset to their default values.

Note: A label is not allowed with this directive. See also PAGE**.**

Example C-57. LSTCOL Directive

```
LSTCOL     40,,,,,20,20; Reset label, X, and Y data field widths
```

C.3.37 MACLIB Macro Library

```
MACLIB     <pathname>
```

This directive is used to specify the <pathname> (as defined by the operating system) of a directory that contains macro definitions. Each macro definition must be in a separate file, and the file must be named the same as the macro with the extension .ASM added. For example, BLOCKMV.ASM would be a file that contained the definition of the macro called BLOCKMV.

If the assembler encounters a directive in the operation field that is not contained in the directive or mnemonic tables, the directory specified by <pathname> is searched for a file of the unknown name (with the .ASM extension added). If such a file is found, the current source line is saved, and the file is opened for input as an INCLUDE file. When the end of the file is encountered, the source line is restored and processing is resumed. Because the

source line is restored, the processed file must have a macro definition of the unknown directive name or else an error will result when the source line is restored and processed. However, the processed file is not limited to macro definitions and can include any legal source code statements.

Multiple MACLIB directives may be given, in which case the assembler will search each directory in the order in which it is encountered.

Note: A label is not allowed with this directive. See also INCLUDE**.**

Example C-58. MACLIB Directive

```
MACLIB  'macros\mymacs\'; IBM PC example
MACLIB  'fftlib/'      ; UNIX example
```

C.3.38 MACRO Macro Definition

```
<label>    MACRO     [<dummy argument list>]
           .
           .
           <macro definition statements>
           .
           .
           ENDM
```

The dummy argument list has the following form

```
[<dumarg>[,<dumarg>,...,<dumarg>]]
```

The required label is the symbol by which the macro is called. If the macro is named the same as an existing assembler directive or mnemonic, a warning is issued. This warning can be avoided with the RDIRECT directive.

The definition of a macro consists of three parts: the header, which assigns a name to the macro and defines the dummy arguments; the body, which consists of prototype or skeleton source statements; and the terminator. The header is the MACRO directive, its label, and the dummy argument list. The body contains the pattern of standard source statements. The terminator is the ENDM directive.

The dummy arguments are symbolic names that the macro processor replaces with arguments when the macro is expanded (called). Each dummy argument must obey the same rules as symbol names. Dummy argument names that are preceded by an underscore are not allowed. Within each of the three dummy argument fields, the dummy arguments are separated by commas. The dummy argument fields are separated by one or more blanks.

Macro definitions may be nested but the nested macro is not defined until the primary macro is expanded.

Note: See also DUP, DUPA, DUPC, DUPF, ENDM.

Example C-59. MACRO Directive

```
SWAP_SYM    MACRO       REG1,REG2  ;swap REG1,REG2 using X0 as temp
            MOVE        R\?REG1,X0
            MOVE        R\?REG2,R\?REG1
            MOVE        X0,R\?REG2
            ENDM
```

C.3.39 MODE Change Relocation Mode

```
            MODE        <ABS[OLUTE] | REL[ATIVE]>
```

The MODE directive causes the assembler to change to the designated operational mode. This directive may be given at any time in the assembly source to alter the set of location counters used for section addressing. Code generated while in absolute mode is placed in memory at the location determined during assembly. Relocatable code and data are based from the enclosing section start address. The MODE directive has no effect when the command line -A option is issued.

Note: A label is not allowed with this directive. See also ORG.

Example C-60. MODE Directive

```
            MODE        ABS         ; Change to absolute mode
```

C.3.40 MSG Programmer Generated Message

```
MSG  [{<str>|<exp>}[,{<str>|<exp>},...,{<str>|<exp>}]]
```

The MSG directive causes a message to be output by the assembler. The error and warning counts are not affected. The MSG directive is normally used in conjunction with conditional assembly directives for informational purposes. The assembly proceeds normally after the message has been printed. An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified optionally to describe the nature of the message.

Note: A label is not allowed with this directive. See also FAIL, WARN.

Example C-61. MSG Directive

```
            MSG         'Generating sine tables'
```

C.3.41 NOLIST Stop Assembly Listing

```
NOLIST
```

Do not print the listing from this point on (including the NOLIST directive). Subsequent source lines will not be printed.

The NOLIST directive actually decrements a counter that is checked for a positive value and is symmetrical with respect to the LIST directive. Note the following sequence:

```
; Counter value currently 1
LIST                  ; Counter value = 2
LIST                  ; Counter value = 3
NOLIST                ; Counter value = 2
NOLIST                ; Counter value = 1
```

The listing still is not disabled until another NOLIST directive is issued.

Note: A label is not allowed with this directive. See also LIST, OPT.

Example C-62. NOLIST Directive

```
IF         LISTOFF
NOLIST                ; Turn the listing off
ENDIF
```

C.3.42 OPT Assembler Options

```
OPT        <option>[,<option>,...,<option>][<comment>]
```

The OPT directive is used to designate the assembler options. Assembler options are given in the operand field of the source input file and are separated by commas. Options also may be specified using the command line -O option. All options have a default condition. Some options are reset to their default condition at the end of pass one. Some are allowed to have the prefix NO attached to them, which then reverses their meaning.

Note: A label is not allowed with this directive.

Options can be grouped by function into five different types:

- Listing format control
- Reporting options
- Message control
- Symbol options
- Assembler operation

C.3.42.1 Listing Format Control

These options control the format of the listing file:

- FC—Fold trailing comments
- FF— Form feeds for page ejects
- FM—Format messages
- PP—Pretty print listing
- RC—Relative comment spacing

C.3.42.2 Reporting Options

These options control what is reported in the listing file:

- CEX—Print DC expansions
- CL—Print conditional assembly directives
- CRE—Print symbol cross-reference
- DXL—Expand DEFINE directive strings in listing
- HDR—Generate listing headers
- IL—Inhibit source listing
- LOC—Print local labels in cross-reference
- MC—Print macro calls
- MD—Print macro definitions
- MEX—Print macro expansions
- MU—Print memory utilization report
- NL—Print conditional assembly and section nesting levels
- S—Print symbol table
- U—Print skipped conditional assembly lines

C.3.42.3 Message Control

These options control the types of assembler messages that are generated:

- AE—Check address expressions
- MSW—Warn on memory space incompatibilities
- UR—Flag unresolved references
- W—Display warning messages

C.3.42.4 Symbol Options

These options deal with the handling of symbols by the assembler:

- DEX—Expand DEFINE symbols within quoted strings
- IC—Ignore case in symbol names
- NS—Support symbol scoping in nested sections
- SCL—Scope structured control statement labels
- SCO—Structured control statement labels to listing / object file
- SO—Write symbols to object file
- XLL—Write local labels to object file
- XR—Recognize XDEFed symbols without XREF

C.3.42.5 Assembler Operation

These are miscellaneous options having to do with internal assembler operation:

- CC—Enable cycle counts
- CK—Enable checksumming
- CM—Preserve comment lines within macros
- CONST—Make EQU symbols assembly time constants
- CONTCK—Continue checksumming
- DLD—Do not restrict directives in loops
- GL—Make all section symbols global
- GS—Make all sections global static
- INTR—Perform interrupt location checks
- LB—Byte increment load counter
- LDB—Listing file debug
- MI—Scan MACLIB directories for include files
- PS—Pack strings
- PSM—Programmable short addressing mode
- RP—Generate NOP to accommodate pipeline delay
- RSV—Check reserve data memory locations
- SI—Interpret short immediate as long or sign extended
- SVO—Preserve object file on errors

Following are descriptions of the individual options. The parenthetical inserts specify default if the option is the default condition and reset if the option is reset to its default state at the end of pass one.

- **AE—**(default, reset) Check address expressions for appropriate arithmetic operations. For example, this will check that only valid add or subtract operations are performed on address terms.
- **CC—**Enable cycle counts and clear total cycle count. Cycle counts are shown on the output listing for each instruction. Cycle counts assume a full instruction fetch pipeline and no wait states.
- CEX—Print DC expansions.
- CK—Enable checksumming of instruction and data values and clear cumulative checksum. The checksum value can be obtained using the @CHK() function.
- CL—(default, reset) Print the conditional assembly directives.
- CM—(default, reset) Preserve comment lines of macros when they are defined. Note that any comment line within a macro definition that starts with two consecutive semicolons (;;) is never preserved in the macro definition.
- CONST—EQU symbols are maintained as assembly time constants and will not be sent to the object file. This option, if used, must be specified before the first symbol in the source program is defined.
- CONTC—Reenable cycle counts. Does not clear total cycle counts. The cycle count for each instruction is shown on the output listing.
- CONTCK—Reenable checksumming of instructions and data. Does not clear cumulative checksum value.
- CRE—Print a cross reference table at the end of the source listing. This option, if used, must be specified before the first symbol in the source program is defined.
- DEX—Expand DEFINE symbols within quoted strings. Can also be done on a case-by-case basis using double-quoted strings.
- DLD—Do not restrict directives in DO loops. The presence of some directives in DO loops does not make sense, including some OPT directive variations. This option suppresses errors on particular directives in loops.
- DXL—(default, reset) Expand DEFINE directive strings in listing.
- FC—Fold trailing comments. Any trailing comments that are included in a source line are folded underneath the source line and aligned with the opcode field. Lines that start with the comment character are aligned with the label field in the source listing. The FC option is useful for displaying the source listing on 80 column devices.
- FF—Use form feeds for page ejects in the listing file.
- FM—Format assembler messages so that the message text is aligned and broken at word boundaries.

- GL—Make all section symbols global. This has the same effect as declaring every section explicitly GLOBAL. This option must be given before any sections are defined explicitly in the source file.
- GS—(default, reset in absolute mode) Make all sections global static. All section counters and attributes are associated with the GLOBAL section. This option must be given before any sections are defined explicitly in the source file.
- HDR—(default, reset) Generate listing header along with titles and subtitles.
- IC—Ignore case in symbol, section, and macro names. This directive must be issued before any symbols, sections, or macros are defined.
- IL—Inhibit source listing. This option will stop the assembler from producing a source listing.
- INTR—(default, reset in absolute mode) Perform interrupt location checks. Certain DSP instructions may not appear in the interrupt vector locations in program memory. This option enables the assembler to check for these instructions when the program counter is within the interrupt vector bounds.
- LB—Increment load counter (if different from runtime) by number of bytes in DSP word to provide byte-wide support for overlays in bootstrap mode. This option must appear before any code or data generation.
- LDB—Use the listing file as the debug source file rather than the assembly language file. The -L command line option to generate a listing file must be specified for this option to take effect.
- LOC—Include local labels in the symbol table and cross-reference listing. Local labels are not normally included in these listings. If neither the S or CRE options are specified, then this option has no effect. The LOC option must be specified before the first symbol is encountered in the source file.
- MC—(default, reset) Print macro calls.
- MD—(default, reset) Print macro definitions.
- MEX—Print macro expansions.
- MI—Scan MACLIB directory paths for include files. The assembler ordinarily looks for included files only in the directory specified in the INCLUDE directory or in the paths given by the -I command line option. If the MI option is used the assembler also looks for included files in any designated MACLIB directories.
- MSW—(default, reset) Issue warning on memory space incompatibilities.
- MU—Include a memory utilization report in the source listing. This option must appear before any code or data generation.
- NL—Display conditional assembly (IF-ELSE-ENDIF) and section nesting levels on listing.
- NOAE—Do not check address expressions.
- NOCC—(default, reset) Disable cycle counts. Does not clear total cycle count.
- NOCEX—(default, reset) Do not print DC expansions.

- NOCK—(default, reset) Disable checksumming of instruction and data values.
- NOCL—Do not print the conditional assembly directives.
- NOCM—Do not preserve comment lines of macros when they are defined.
- NODEX—(default, reset) Do not expand DEFINE symbols within quoted strings.
- NODLD—(default, reset) Restrict use of certain directives in DO loop.
- NODXL—Do not expand DEFINE directive strings in listing.
- NOFC—(default, reset) Inhibit folded comments.
- NOFF—(default, reset) Use multiple line feeds for page ejects in the listing file.
- NOFM—(default, reset) Do not format assembler messages.
- NOGS—(default, reset in relative mode) Do not make all sections global static.
- NOHDR—Do not generate listing header. This also turns off titles and subtitles.
- NOINTR—(default, reset in relative mode) Do not perform interrupt location checks.
- NOMC—Do not print macro calls.
- NOMD—Do not print macro definitions.
- NOMEX—(default, reset) Do not print macro expansions.
- NOMI—(default, reset) Do not scan MACLIB directory paths for include files.
- NOMSW—Do not issue warning on memory space incompatibilities.
- NONL—(default, reset) Do not display nesting levels on listing.
- NONS—Do not allow scoping of symbols within nested sections.
- NOPP—Do not pretty print listing file. Source lines are sent to the listing file as they are encountered in the source, with the exception that tabs are expanded to spaces and continuation lines are concatenated into a single physical line for printing.
- NOPS—Do not pack strings in DC directive. Individual bytes in strings are stored one byte per word.
- NORC—(default, reset) Do not space comments relatively.
- NORP—(default, reset) Do not generate instructions to accommodate pipeline delay.
- NOSCL—Do not maintain the current local label scope when a structured control statement label is encountered.
- NOU—(default, reset) Do not print the lines excluded from the assembly due to a conditional assembly directive.
- NOUR—(default, reset) Do not flag unresolved external references.
- NOW—Do not print warning messages.
- NS—(default, reset) Allow scoping of symbols within nested sections.

- PP—(default, reset) Pretty print listing file. The assembler attempts to align fields at a consistent column position without regard to source file formatting.
- PS—(default, reset) Pack strings in DC directive. Individual bytes in strings are packed into consecutive target words for the length of the string.
- RC—Space comments relatively in listing fields. By default, the assembler always places comments at a consistent column position in the listing file. This option allows the comment field to float: on a line containing only a label and opcode, the comment begins in the operand field.
- RP—Generate NOP instructions to accommodate pipeline delay. If an address register is loaded in one instruction then the contents of the register is not available for use as a pointer until <u>after</u> the next instruction. Ordinarily when the assembler detects this condition it issues an error message. The RP option will cause the assembler to output a NOP instruction into the output stream instead of issuing an error.
- S—Print symbol table at the end of the source listing. This option has no effect if the CRE option is used.
- SCL—(default, reset) Structured control statements generate non-local labels that ordinarily are not visible to the programmer. This can create problems when local labels are interspersed among structured control statements. This option causes the assembler to maintain the current local label scope when a structured control statement label is encountered.
- SCO—Send structured control statement labels to object and listing files. Normally the assembler does not externalize these labels. This option must appear before any symbol definition.
- SO—Write symbol information to object file. This option is recognized but performs no operation in COFF assemblers.
- SVO—Preserve object file on errors. Normally any object file produced by the assembler is deleted if errors occur during assembly. This option must be given before any code or data is generated.
- U—Print the unassembled lines skipped due to failure to satisfy the condition of a conditional assembly directive.
- UR—Generate a warning at assembly time for each unresolved external reference. This option works only in relocatable mode.
- W—(default, reset) Print all warning messages.
- WEX—Add warning count to exit status. Ordinarily the assembler exits with a count of errors. This option causes the count of warnings to be added to the error count.
- XLL—Write underscore local labels to object file. This is primarily used to aid debugging. This option, if used, must be specified before the first symbol in the source program is defined.

- XR—Causes XDEFed symbols to be recognized within other sections without being XREFed. This option, if used, must be specified before the first symbol in the source program is encountered.

Example C-63. OPT Directive

```
          OPT        CEX,MEX    ; Turn on DC and macro expansions
          OPT        CRE,MU     ; Cross reference, memory utilization
```

C.3.43 ORG Initialize Memory Space and Location Counters

```
ORG  <rms>[<rlc>][<rmp>]:[<exp1>][,<lms>[<llc>][<lmp>]:[<exp2>]]
ORG  <rms>[<rmp>][(<rce>)]:[<exp1>][,<lms>[<lmp>][(<lce>)]:[<exp2>]]
```

The ORG directive is used to specify addresses and to indicate memory space and mapping changes. It also can designate an implicit counter mode switch in the assembler and serves as a mechanism for initiating overlays.

Note: A label is not allowed with this directive.

The parameters used with the ORG directive are as follows

- <rms>—Which memory space (X, Y, L, P, or E) is used as the runtime memory space. If the memory space is L, any allocated datum with a value greater than the target word size is extended to two words; otherwise, it is truncated. If the memory space is E, then depending on the memory space qualifier, any generated words are split into bytes, one byte per word, or a 16/8-bit combination.
- <rlc>—Which runtime counter H, L, or default (if neither H or L is specified), that is associated with the <rms> is used as the runtime location counter.
- <rmp>—Indicates the runtime physical mapping to DSP memory: I—internal, E—external, R—ROM, A—port A, B—port B. If not present, no explicit mapping is done.
- <rce>—Non-negative absolute integer expression representing the counter number to be used as the runtime location counter. Must be enclosed in parentheses. Should not exceed the value 65535.
- <exp1>—Initial value to assign to the runtime counter used as the <rlc>. If <exp1> is a relative expression the assembler uses the relative location counter. If <exp1> is an absolute expression the assembler uses the absolute location counter. If <exp1> is not specified, then the last value and mode that the counter had is used.
- <lms>—Which memory space (X, Y, L, P, or E) is used as the load memory space. If the memory space is L, any allocated datum with a value greater than the target word size is extended to two words; otherwise, it is truncated. If the memory space is E, then depending on the memory space qualifier, any generated words are split into bytes, one byte per word, or a 16/8-bit combination.

- <llc>—Which load counter, H, L, or default (if neither H or L is specified), that is associated with the <lms> is used as the load location counter.
- <lmp>—Indicates the load physical mapping to DSP memory: I—internal, E—external, R—ROM, A—port A, B—port B. If not present, no explicit mapping is done.
- <lce>—Non-negative absolute integer expression representing the counter number to be used as the load location counter. Must be enclosed in parentheses. Should not exceed the value 65535.
- <exp2>—Initial value to assign to the load counter used as the <llc>. If <exp2> is a relative expression the assembler uses the relative location counter. If <exp2> is an absolute expression the assembler uses the absolute location counter. If <exp2> is not specified, then the last value and mode that the counter had is used.

If the last half of the operand field in an ORG directive dealing with the load memory space and counter is not specified, then the assembler assumes that the load memory space and load location counter are the same as the runtime memory space and runtime location counter. In this case, object code is being assembled to be loaded into the address and memory space where it is when the program is run; it is not an overlay.

If the load memory space and counter are given in the operand field, then the assembler always generates code for an overlay. Whether the overlay is absolute or relocatable depends upon the current operating mode of the assembler and whether the load counter value is an absolute or relative expression. If the assembler is running in absolute mode, or if the load counter expression is absolute, then the overlay is absolute. If the assembler is in relative mode and the load counter expression is relative, the overlay is relocatable. Runtime relocatable overlay code is addressed relative to the location given in the runtime location counter expression. This expression, if relative, may not refer to another overlay block.

Note: See also MODE.

Example C-64. ORG Directive

```
ORG P:$1000
```

Sets the runtime memory space to P. Selects the default runtime counter (counter 0) associated with P space to use as the runtime location counter and initializes it to $1000. The load memory space is implied to be P, and the load location counter is assumed to be the same as the runtime location counter.

Example C-64. ORG Directive (Continued)

`ORG  PHE:`

Sets the runtime memory space to P. Selects the H load counter (counter 2) associated with P space to use as the runtime location counter. The H counter will not be initialized, and its last value is used. Code generated hereafter is mapped to external (E) memory. The load memory space is implied to be P, and the load location counter is assumed to be the same as the runtime location counter.

`ORG  PI:OVL1,Y:`

Indicates code is generated for an overlay. The runtime memory space is P, and the default counter is used as the runtime location counter. It is reset to the value of OVL1. If the assembler is in absolute mode via the -A command line option then OVL1 must be an absolute expression. If OVL1 is an absolute expression the assembler uses the absolute runtime location counter. If OVL1 is a relocatable value the assembler uses the relative runtime location counter. In this case OVL1 must not itself be an overlay symbol (i.e., defined within an overlay block). The load memory space is Y. Since neither H, L, nor any counter expression was specified as the load counter, the default load counter (counter 0) is used as the load location counter. The counter value and mode is whatever it was the last time it was referenced.

`ORG  XL:,E8:`

Sets the runtime memory space to X. Selects the L counter (counter 1) associated with X space to use as the runtime location counter. The L counter is not initialized, and its last value is used. The load memory space is set to E, and the qualifier 8 indicates a bytewise RAM configuration. Instructions and data are generated eight bits per output word with byte-oriented load addresses. The default load counter is used, and there is no explicit load origin.

`ORG  P(5):,Y:$8000`

Indicates code is generated for an absolute overlay. The runtime memory space is P, and the counter used as the runtime location counter is counter 5. It will not be initialized, and the last previous value of counter 5 is used. The load memory space is Y. Since neither H, L, nor any counter expression was specified as the load counter, the default load counter (counter 0) is used as the load location counter. The default load counter is initialized to $8000.

C.3.44 PAGE Top of Page/Size Page

```
PAGE            [<exp1>[,<exp2>...,<exp5>]]
```

The PAGE directive has two forms:

1. If no arguments are supplied, then the assembler advances the listing to the top of the next page. In this case, the PAGE directive is not output.
2. The PAGE directive with arguments can be used to specify the printed format of the output listing. Arguments may be any positive absolute integer expression. The arguments in the operand field (as explained below) are separated by commas. Any argument can be left as the default or last set value by omitting the argument and using two adjacent commas. The PAGE directive with arguments will not cause a page eject and is printed in the source listing.

Note: A label is not allowed with this directive.

The arguments in order are as follows:

1. **PAGE_WIDTH <exp1>—**Page width in terms of number of output columns per line (default 80, min 1, max 255).
2. **PAGE_LENGTH <exp2>—**Page length in terms of total number of lines per page (default 66, min 10, max 255). As a special case a page length of zero turns off all headers, titles, subtitles, and page breaks.
3. **BLANK_TOP <exp3>**—Blank lines at top of page (default 0, min 0, max see below).
4. **BLANK_BOTTOM <exp4>—**Blank lines at bottom of page (default 0, min 0, max see below).
5. **BLANK_LEFT <exp5>—**Blank left margin. Number of blank columns at the left of the page (default 0, min 0, max see below).

The following relationships must be maintained:

```
BLANK_TOP + BLANK_BOTTOM <= PAGE_LENGTH - 10
BLANK_LEFT < PAGE_WIDTH
```

Note: See also LSTCOL.

Example C-65. PAGE Directive

```
PAGE        132,,3,3    ; Set width to 132, 3 line top/bottom margins
PAGE                    ; Page eject
```

C.3.45 PMACRO Purge Macro Definition

```
PMACRO     <symbol>[,<symbol>,...,<symbol>]
```

The specified macro definition is purged from the macro table, allowing the macro table space to be reclaimed.

Note: A label is not allowed with this directive. See also MACRO**.**

Example C-66. PMACRO Directive

```
          PMACRO      MAC1,MAC2
```

This statement would cause the macros named MAC1 and MAC2 to be purged.

C.3.46 PRCTL Send Control String to Printer

```
          PRCTL       <exp>I<string>,...,<exp>I<string>
```

PRCTL simply concatenates its arguments and ships them to the listing file. (The directive line itself is not printed unless there is an error.) <exp> is a byte expression and <string> is an assembler string. A byte expression would be used to encode non-printing control characters, such as ESC. The string may be of arbitrary length, up to the maximum assembler-defined limits.

PRCTL may appear anywhere in the source file and the control string is output at the corresponding place in the listing file. However, if a PRCTL directive is the last line in the last input file to be processed, the assembler insures that all error summaries, symbol tables, and cross-references have been printed before sending out the control string. This is so a PRCTL directive can be used to restore a printer to a previous mode after printing is done. Similarly, if the PRCTL directive appears as the first line in the first input file, the control string is output before page headings or titles.

The PRCTL directive only works if the -L command line option is given; otherwise it is ignored.

Note: A label is not allowed with this directive.

Example C-67. PRCTL Directive

```
          PRCTL       $1B,'E'    ; Reset HP LaserJet printer
```

C.3.47 RADIX Change Input Radix for Constants

```
          RADIX       <expression>
```

Changes the input base of constants to the result of <expression>. The absolute integer expression must evaluate to one of the legal constant bases (2, 10, or 16). The default radix is 10. The RADIX directive allows the programmer to specify constants in a preferred radix without a leading radix indicator. The radix prefix for base 10 numbers is the grave accent (**‘**). Note that if a constant is used to alter the radix, it must be in the appropriate input base at the time the RADIX directive is encountered.

Note: A label is not allowed with this directive.

Example C-68. RADIX Directive

```
_RAD10    DC      10        ; Evaluates to hex A
          RADIX   2
_RAD2     DC      10        ; Evaluates to hex 2
          RADIX   ‘16
_RAD16    DC      10        ; Evaluates to hex 10
          RADIX   3         ; Bad radix expression
```

C.3.48 RDIRECT Remove Directive or Mnemonic from Table

```
          RDIRECT   <direc>[,<direc>,...,<direc>]
```

The RDIRECT directive is used to remove directives from the assembler directive and mnemonic tables. If the directive or mnemonic that has been removed is later encountered in the source file, it is assumed to be a macro. Macro definitions that have the same name as assembler directives or mnemonics will cause a warning message to be output unless the RDIRECT directive has been used to remove the directive or mnemonic name from the assembler’s tables. Additionally, if a macro is defined through the MACLIB directive which has the same name as an existing directive or opcode, it will not automatically replace that directive or opcode as previously described. In this case, the RDIRECT directive must be used to force the replacement.

Since the effect of this directive is global, it cannot be used in an explicitly-defined section (see SECTION directive). An error results if the RDIRECT directive is encountered in a section.

Note: A label is not allowed with this directive.

Example C-69. RDIRECT Directive

```
          RDIRECT   PAGE,MOVE
```

This causes the assembler to remove the PAGE directive from the directive table and the MOVE mnemonic from the mnemonic table.

C.3.49 SCSJMP Set Structured Control Statement Branching Mode

```
          SCSJMP    {SHORT | LONG | NONE}
```

The SCSJMP directive is analogous to the FORCE directive, but it only applies to branches generated automatically by structured control statements. (See **Section C.4** on page C-53.) There is no explicit way, as with a forcing operator, to force a branch short or long when it is produced by a structured control statement. This directive causes all branches resulting from subsequent structured control statements to be forced to the specified mode.

Just like the FORCE pseudo-op, errors can result if a value is too large to be forced short. For relocatable code, the error may not occur until the linking phase.

Note: See also FORCE, SCSREG**.** A label is not allowed with this directive.

Example C-70. SCSJMP Directive

```
        SCSJMP    SHORT     ; force all subsequent SCS jumps short
```

C.3.50 SCSREG Reassign Structured Control Statement Registers

```
SCSREG    [<srcreg>[,<dstreg>[,<tmpreg>[,<extreg>]]]]
```

The SCSREG directive reassigns the registers used by structured control statement (SCS) directives. It is convenient for reclaiming default SCS registers when they are needed as application operands within a structured control construct. <srcreg> is ordinarily the source register for SCS data moves. <dstreg> is the destination register. <tmpreg> is a temporary register for swapping SCS operands. <extreg> is an extra register for complex SCS operations. With no arguments, SCSREG resets the SCS registers to their default assignments.

The SCSREG directive should be used judiciously to avoid register context errors during SCS expansion. Source and destination registers may not necessarily be used strictly as source and destination operands. The assembler does no checking of reassigned registers beyond validity for the target processor. Errors can result when a structured control statement is expanded and an improper register reassignment has occurred. It is recommended that the MEX option (see the OPT directive) be used to examine structured control statement expansion for relevant constructs to determine default register usage and applicable reassignment strategies.

Note: See also OPT (MEX), SCSJMP**.** A label is not allowed with this directive.

Example C-71. SCSREG Directive

```
    SCSREG    Y0,B    ; reassign SCS source and dest. registers
```

C.3.51 SECTION Start Section

```
SECTION    <symbol>   [GLOBAL | STATIC | LOCAL]
.
.
<section source statements>
.
.
ENDSEC
```

The SECTION directive defines the start of a section. All symbols that are defined within a section have the <symbol> associated with them as their section name. This serves to

protect them from like-named symbols elsewhere in the program. By default, a symbol defined inside any given section is private to that section unless the GLOBAL or LOCAL qualifier accompanies the SECTION directive.

Any code or data inside a section is considered an indivisible block with respect to relocation. Code or data associated with a section is independently relocatable within the memory space to which it is bound, unless the STATIC qualifier follows the SECTION directive on the instruction line.

Symbols within a section are generally distinct from other symbols used elsewhere in the source program, even if the symbol name is the same. This is true as long as the section name associated with each symbol is unique, the symbol is not declared public (XDEF/GLOBAL), and the GLOBAL or LOCAL qualifier is not used in the section declaration. Symbols that are defined outside of a section are considered global symbols and have no explicit section name associated with them. Global symbols may be referenced freely from inside or outside of any section, as long as the global symbol name does not conflict with another symbol by the same name in a given section.

If the GLOBAL qualifier follows the <section name> in the SECTION directive, then all symbols defined in the section until the next ENDSEC directive are considered global. The effect is as if every symbol in the section were declared with GLOBAL. This is useful when a section needs to be independently relocatable, but data hiding is not desired.

If the STATIC qualifier follows the <section name> in the SECTION directive, then all code and data defined in the section until the next ENDSEC directive are relocated in terms of the immediately enclosing section. The effect with respect to relocation is as if all code and data in the section were defined within the parent section. This is useful when a section needs data hiding, but independent relocation is not required.

If the LOCAL qualifier follows the <section name> in the SECTION directive, then all symbols defined in the section until the next ENDSEC directive are visible to the immediately enclosing section. The effect is as if every symbol in the section were defined within the parent section. This is useful when a section needs to be independently relocatable, but data hiding within an enclosing section is not required.

The division of a program into sections controls not only labels and symbols but also macros and DEFINE directive symbols. Macros defined within a section are private to that section and are distinct from macros defined in other sections even if they have the same macro name. Macros defined outside of sections are considered global and may be used within any section. Similarly, DEFINE directive symbols defined within a section are private to that section and DEFINE directive symbols defined outside of any section are globally applied. There are no directives that correspond to XDEF for macros or DEFINE

symbols, and therefore, macros and DEFINE symbols defined in a section can never be accessed globally. If global accessibility is desired, the macros and DEFINE symbols should be defined outside of any section.

Sections can be nested to any level. When the assembler encounters a nested section, the current section is stacked and the new section is used. When the ENDSEC directive of the nested section is encountered, the assembler restores the old section and uses it. The ENDSEC directive always applies to the most previous SECTION directive. Nesting sections provides a measure of scoping for symbol names, in that symbols defined within a given section are visible to other sections nested within it. For example, if section B is nested inside section A, then a symbol defined in section A can be used in section B without XDEFing in section A or XREFing in section B. This scoping behavior can be turned off and on with the NONS and NS options respectively. (See the OPT directive.)

Sections may also be split into separate parts. That is, <section name> can be used multiple times with SECTION and ENDSEC directive pairs. If this occurs, then these separate (but identically named) sections can access each others symbols freely without the use of the XREF and XDEF directives. If the XDEF and XREF directives are used within one section, they apply to all sections with the same section name. The reuse of the section name is allowed to permit the program source to be arranged in an arbitrary manner (for example, all statements that reserve X space storage locations grouped together) but retain the privacy of the symbols for each section.

When the assembler operates in relative mode (the default), sections act as the basic grouping for relocation of code and data blocks. For every section defined in the source, a set of location counters is allocated for each DSP memory space. These counters are used to maintain offsets of data and instructions relative to the beginning of the section. At link time, sections can be relocated to an absolute address, loaded in a particular order, or linked contiguously as specified by the programmer. Sections which are split into parts or among files are logically recombined so that each section can be relocated as a unit.

Sections may be relocatable or absolute. In the assembler absolute mode (command line -A option) all sections are considered absolute. A full set of locations counters is reserved for each absolute section unless the GS option is given. (See the OPT directive.) In relative mode, all sections are initially relocatable. However, a section or a part of a section may be made absolute either implicitly by using the ORG directive or explicitly through use of the MODE directive.

Note: A label is not allowed with this directive. See also MODE, ORG, GLOBAL, LOCAL, XDEF, XREF.

Example C-72. SECTION Directive

```
          SECTION    TABLES     ; TABLES will be the section name
```

C.3.52 SET Set Symbol to a Value

```
          <label>    SET        <expression>
                     SET        <label>    <expression>
```

The SET directive is used to assign the value of the expression in the operand field to the label. The SET directive functions somewhat like the EQU directive. However, labels defined via the SET directive can have their values redefined in another part of the program (but only through the use of another SET directive). The SET directive is useful in establishing temporary or reusable counters within macros. The expression in the operand field of a SET must be absolute and cannot include a symbol that is not yet defined. (No forward references are allowed.)

Note: See also EQU, GSET.

Example C-73. SET Directive

```
COUNT     SET        0          ; INITIALIZE COUNT
```

C.3.53 STITLE Initialize Program Sub-Title

```
          STITLE     [<string>]
```

The STITLE directive initializes the program subtitle to the string in the operand field. The subtitle is printed on the top of all succeeding pages until another STITLE directive is encountered. The subtitle is initially blank. The STITLE directive will not be printed in the source listing. An STITLE directive with no string argument causes the current subtitle to be blank.

Note: A label is not allowed with this directive. See also TITLE**.**

Example C-74. STITLE Directive

```
          STITLE     'COLLECT SAMPLES'
```

C.3.54 SYMOBJ Write Symbol Information to Object File

```
          SYMOBJ     <symbol>[,<symbol>,...,<symbol>]
```

The SYMOBJ directive causes information for each <symbol> to be written to the object file. This directive is recognized but currently performs no operation in COFF assemblers.

Note: A label is not allowed with this directive.

Example C-75. SYMOBJ

```
        SYMOBJ      XSTART,HIRTN,ERRPROC
```

C.3.55 TABS Set Listing Tab Stops

```
        TABS        <tabstops>
```

The TABS directive allows resetting the listing file tab stops from the default value of 8.

Note: A label is not allowed with this directive. See also LSTCOL.

Example C-76. TABS Directive

```
        TABS        4          ; Set listing file tab stops to 4
```

C.3.56 TITLE Initialize Program Title

```
        TITLE       [<string>]
```

The TITLE directive initializes the program title to the string in the operand field. The program title is printed on the top of all succeeding pages until another TITLE directive is encountered. The title is initially blank. The TITLE directive is not printed in the source listing. A TITLE directive with no string argument causes the current title to be blank.

Note: A label is not allowed with this directive. See also STITLE.

Example C-77. TITLE Directive

```
        TITLE       'FIR FILTER'
```

C.3.57 UNDEF Undefine DEFINE Symbol

```
        UNDEF       [<symbol>]
```

The UNDEF directive causes the substitution string associated with <symbol> to be released, and <symbol> will no longer represent a valid DEFINE substitution. See the DEFINE directive for more information.

Note: A label is not allowed with this directive. See also DEFINE.

Example C-78. UNDEF Directive

```
        UNDEF       DEBUG ; UNDEFINES THE DEBUG SUBSTITUTION STRING
```

C.3.58 WARN Programmer Generated Warning

```
      WARN    [{<str>|<exp>}[,{<str>|<exp>},...,{<str>|<exp>}]]
```

The WARN directive causes a warning message to be output by the assembler. The total warning count is incremented as with any other warning. The WARN directive is

normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the warning has been printed. An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified optionally to describe the nature of the generated warning.

Note: A label is not allowed with this directive. See also FAIL, MSG.

Example C-79. WARN Directive

```
        WARN        'parameter too large'
```

C.3.59 XDEF External Section Symbol Definition

```
        XDEF          <symbol>[,<symbol>,...,<symbol>]
```

The XDEF directive is used to specify that the list of symbols is defined within the current section, and that those definitions should be accessible by sections with a corresponding XREF directive. This directive is only valid if used within a program section bounded by the SECTION and ENDSEC directives. The XDEF directive must appear before <symbol> is defined in the section. If the symbols that appear in the operand field are not defined in the section, an error is generated.

Note: A label is not allowed with this directive. See also SECTION, XREF.

Example C-80. XDEF Directive

```
  SECTION    IO
  XDEF       LOOPA   ; LOOPA will be accessible by sections with XREF
  .
  .
  .
  ENDSEC
```

C.3.60 XREF External Section Symbol Reference

```
        XREF          <symbol>[,<symbol>,...,<symbol>]
```

The XREF directive is used to specify that the list of symbols is referenced in the current section but is not defined within the current section. These symbols must either have been defined outside of any section or declared as globally accessible within another section using the XDEF directive. If the XREF directive is not used to specify that a symbol is defined globally and the symbol is not defined within the current section, an error is generated, and all references within the current section to such a symbol are flagged as undefined. The XREF directive must appear before any reference to <symbol> in the section.

Note: A label is not allowed with this directive. See also SECTION, XDEF.

Example C-81. XREF Directive

```
        SECTION    FILTER
        XREF       AA,CC,DD   ; XDEFed symbols within section
        .
        .
        .
        ENDSEC
```

C.4 Structured Control Statements

An assembly language provides an instruction set for performing certain rudimentary operations. These operations in turn may be combined into control structures such as loops (FOR, REPEAT, WHILE) or conditional branches (IF-THEN, IF-THEN-ELSE). The assembler, however, accepts formal, high-level directives that specify these control structures, generating the appropriate assembly language instructions for their efficient implementation. This use of structured control statement directives improves the readability of assembly language programs without compromising the desirable aspects of programming in an assembly language.

C.4.1 Structured Control Directives

The following directives are used for structured control. Note the leading period, which distinguishes these keywords from other directives and mnemonics. Structured control directives may be specified in either upper or lower case, but they must appear in the opcode field of the instruction line (i.e., they must be preceded either by a label, a space, or a tab).

.BREAK	**.ENDI**	**.LOOP**
.CONTINUE	**.ENDL**	**.REPEAT**
.ELSE	**.ENDW**	**.UNTIL**
.ENDF	**.FOR**	**.WHILE**
.IF		

In addition, the following keywords are used in structured control statements:

AND	**DOWNTO**	**TO**
BY	**OR**	
DO	**THEN**	

Note: AND, DO, and OR are reserved assembler instruction mnemonics.

C.4.2 Syntax

The formats for the .BREAK, .CONTINUE, .FOR, .IF, .LOOP, .REPEAT, and .WHILE statements are given in sections C.4.2.1 through C.4.2.7. Syntactic variables used in the formats are defined as follows:

- **<expression>—**A simple or compound expression (Section C.4.3).
- **<stmtlist>—**Zero or more assembler directives, structured control statements, or executable instructions.

Note: An assembler directive occurring within a structured control statement is examined exactly once—at assembly time. Thus the presence of a directive within a .FOR, .LOOP, .REPEAT, or .WHILE statement does not imply repeated occurrence of an assembler directive; nor does the presence of a directive within an .IF-THEN-.ELSE structured control statement imply conditional assembly.

- **<op1>—**A user-defined operand whose register / memory location holds the .FOR loop counter. The effective address must use a memory alterable addressing mode (i.e., it cannot be an immediate value).
- **<op2>—**The initial value of the .FOR loop counter. The effective address may be any mode and may represent an arbitrary assembler expression.
- **<op3>—**The terminating value of the .FOR loop counter. The effective address may be any mode and may represent an arbitrary assembler expression.
- **<op4>—**The step (increment / decrement) of the .FOR loop counter each time through the loop. If not specified, it defaults to a value of #1. The effective address may be any mode and may represent an arbitrary assembler expression.
- **<cnt>—**The terminating value in a .LOOP statement. This can be any arbitrary assembler expression.

All structured control statements may be followed by normal assembler comments on the same logical line.

C.4.2.1 .BREAK Statement

```
.BREAK
```

The .BREAK statement causes an immediate exit from the innermost enclosing loop construct (.WHILE, .REPEAT, .FOR, .LOOP). A .BREAK statement does not exit an .IF-THEN-.ELSE construct. If a .BREAK is encountered with no loop statement active, a warning is issued.

Note: .BREAK should be used with care near .ENDL directives or near the end of DO loops. It generates a jump instruction which is illegal in those contexts.

Example C-82. .BREAK Statement

```
.WHILE     x:(r1)+ <GT> #0;loop until zero is found
.
.
.
.IF        <cs>
.BREAK                    ;causes exit from WHILE loop
.ENDI
.
.                         ;any instructions here are skipped
.
.ENDW
                          ;execution resumes here after .BREAK
```

C.4.2.2 .CONTINUE Statement

```
.CONTINUE
```

The .CONTINUE statement causes the next iteration of a looping construct (.WHILE, .REPEAT, .FOR, .LOOP) to begin. This means that the loop expression or operand comparison is performed immediately, bypassing any subsequent instructions. If a .CONTINUE is encountered with no loop statement active, a warning is issued.

Note: .CONTINUE should be used with care near .ENDL directives or near the end of DO loops. It generates a jump instruction which is illegal in those contexts. One or more .CONTINUE directives inside a .LOOP construct will generate a NOP instruction just before the loop address.

Example C-83. .CONTINUE Statement

```
.REPEAT
.
.
.
.IF        <cs>
.CONTINUE                 ;causes immediate jump to .UNTIL
.ENDI
.
.                         ;any instructions here are skipped
.
.UNTIL     x:(r1)+ <EQ> #0;evaluation here after .CONTINUE
```

C.4.2.3 .FOR Statement

```
.FOR <op1> = <op2>  {TO | DOWNTO} <op3> [BY <op4>] [DO]
<stmtlist>
.ENDF
```

Initialize <op1> to <op2> and perform <stmtlist> until <op1> is greater (TO) or less than (DOWNTO) <op3>. Makes use of a user-defined operand, <op1>, to serve as a loop counter. .FOR-TO allows counting upward, while .FOR-DOWNTO allows counting

downward. The programmer may specify an increment/decrement step size in <op4>, or elect the default step size of #1 by omitting the BY clause. A .FOR-TO loop is not executed if <op2> is greater than <op3> upon entry to the loop. Similarly, a .FOR-DOWNTO loop is not executed if <op2> is less than <op3>.

<op1> must be a writable register or memory location. It is initialized at the beginning of the loop and updated at each pass through the loop. Any immediate operands must be preceded by a pound sign (#). Memory references must be preceded by a memory space qualifier (X:, Y:, or P:). L memory references are not allowed. Operands must be or refer to single-word values.

The logic generated by the .FOR directive makes use of several DSP data registers. In fact, two data registers are used to hold the step and target values, respectively, throughout the loop; they are never reloaded by the generated code. It is recommended that these registers not be used within the body of the loop, or that they be saved and restored prior to loop evaluation.

Note: The DO keyword is optional.

Example C-84. .FOR Statement

```
.FOR      X:CNT  =  #0  TO  Y:(targ*2)+114; loop on X:CNT
.
.
.
.ENDF
```

C.4.2.4 .IF Statement

```
.IF       <expression>[THEN]
<stmtlist>
[.ELSE
<stmtlist>]
.ENDI
```

If <expression> is true, execute <stmtlist> following THEN (the keyword THEN is optional); if <expression> is false, execute <stmtlist> following .ELSE, if present; otherwise, advance to the instruction following .ENDI.

Note: In the case of nested .IF-THEN-.ELSE statements, each .ELSE refers to the most recent .IF-THEN sequence.

Example C-85. .IF Statement

```
.IF       <EQ>      ; zero bit set?
.
.
.
.ENDI
```

C.4.2.5 .LOOP Statement

```
.LOOP  <cnt>
<stmtlist>
.ENDL
```

Execute <stmtlist> <cnt> times. This is similar to the .FOR loop construct, except that the initial counter and step value are implied to be #1. It is actually a shorthand method for setting up a hardware DO loop on the DSP without having to worry about addressing modes or label placement.

Since the .LOOP statement generates instructions for a hardware DO loop, the same restrictions apply as to the use of certain instructions near the end of the loop, nesting restrictions, etc. One or more .CONTINUE directives inside a .LOOP construct generate a NOP instruction just before the loop address.

Example C-86. .LOOP Statement

```
.LOOP      LPCNT      ; hardware loop LPCNT times
.
.
.
.ENDL
```

C.4.2.6 .REPEAT Statement

```
.REPEAT
<stmtlist>
.UNTIL  <expression>
```

<stmtlist> is executed repeatedly until <expression> is true. When expression becomes true, advance to the next instruction following .UNTIL. The <stmtlist> is executed at least once, even if <expression> is true upon entry to the .REPEAT loop.

Example C-87. .REPEAT Statement

```
.REPEAT
.
.
.
.UNTIL     x:(r1)+ <EQ> #0; loop until zero is found
```

C.4.2.7 .WHILE Statement

```
.WHILE     <expression>[DO]
<stmtlist>
.ENDW
```

The <expression> is tested before execution of <stmtlist>. While <expression> remains true, <stmtlist> is executed repeatedly. When <expression> evaluates false, advance to the

instruction following the .ENDW statement. If <expression> is false upon entry to the .WHILE loop, <stmtlist> is not executed; execution continues after the .ENDW directive.

Note: The DO keyword is optional.

Example C-88. .WHILE Statement

```
            .WHILE     x:(r1)+ <GT> #0; loop until zero is found
            .
            .
            .
            .ENDW
```

C.4.3 Simple and Compound Expressions

Expressions are an integral part of .IF, .REPEAT, and .WHILE statements. Structured control statement expressions should not be confused with the assembler expressions. The latter are evaluated at assembly time and are referred to here as "assembler expressions;" they can serve as operands in structured control statement expressions. The structured control statement expressions described below are evaluated at run time and are referred to in the following discussion simply as “expressions”.

A structured control statement expression may be simple or compound. A compound expression consists of two or more simple expressions joined by either AND or OR (but not both in a single compound expression).

C.4.3.1 Simple Expressions

Simple expressions are concerned with the bits of the condition code register (CCR). These expressions are of two types. The first type merely tests conditions currently specified by the contents of the CCR. (See Section C.4.3.2.) The second type sets up a comparison of two operands to set the condition codes and afterwards tests the codes.

C.4.3.2 Condition Code Expressions

A variety of tests (identical to those in the Jcc instruction) may be performed, based on the CCR condition codes. The condition codes, in this case, are preset by either a user-generated instruction or a structured operand-comparison expression. Each test is expressed in the structured control statement by a mnemonic enclosed in angle brackets.

When processed by the assembler, the expression generates an inverse conditional jump to beyond the matching .ENDx/.UNTIL directive.

Example C-89. Condition Code Expression

```
            .IF         <EQ>        ;zero bit set?
+           bne         Z_L00002    ;code generated by assembler
            CLR         D1          ;user code
            .ENDI
+           Z_L00002                ;assembler-generated label
            .REPEAT                 ;subtract until D0 < D7
+           Z_L00034                ;assembler-generated label
            SUB         D7,D0       ;user code
            .UNTIL      <LT>
+           bge         Z_L00034    ;code generated by assembler
```

C.4.3.3 Operand Comparison Expressions

Two operands may be compared in a simple expression, with subsequent transfer of control based on that comparison. Such a comparison takes the form

```
<op1>  <cc>  <op2>
```

where <cc> is a condition mnemonic enclosed in angle brackets (as described in section C.4.3.2), and <op1> and <op2> are register or memory references, symbols, or assembler expressions. When processed by the assembler, the operands are arranged such that a compare/jump sequence of the following form always results

```
CMP         <reg1>,<reg2>
(J|B)cc     <label>
```

where the jump conditional is the inverse of <cc>. Ordinarily <op1> is moved to the <reg1> data register and <op2> is moved to the <reg2> data register prior to the compare. This is not always the case, however. If <op1> happens to be <reg2> and <op2> is <reg1>, an intermediate register is used as a scratch register. In any event, worstcase code generation for a given operand comparison expression is generally two moves, a compare, and a conditional jump.

Jumps or branches generated by structured control statements are forced long because the number and address of intervening instructions between a control statement and its termination are not known by the assembler. The programmer may circumvent this behavior by use of the SCSJMP directive.

Any immediate operands must be preceded by a pound sign (#). Memory references must be preceded by a memory space qualifier (X:, Y:, or P:). L memory references are not allowed. Operands must be or refer to single-word values.

Note that values in the <reg1> and <reg2> data registers are not saved before expression evaluation. This means that any user data in those registers are overwritten each time the expression is evaluated at runtime. The programmer should take care either to save needed

contents of the registers, reassign data registers using the SCSREG directive, or not use them at all in the body of the particular structured construct being executed.

C.4.3.4 Compound Expressions

A compound expression consists of two or more simple expressions (See Section C.4.3.1.) joined by a logical operator (AND or OR). The boolean value of the compound expression is determined by the boolean values of the simple expressions and the nature of the logical operator. Note that the result of mixing logical operators in a compound expression is undefined:

```
        .IF     X1  <GT>  B  AND  <LS>  AND  R1  <NE>  R2;this is OK
        .IF     X1  <LE>  B  AND  <LC>  OR   R5  <GT>  R6;undefined
```

The simple expressions are evaluated left to right. Note that this means the result of one simple expression could have an impact on the result of subsequent simple expressions, because of the condition code settings stemming from the assembler-generated compare.

If the compound expression is an AND expression and one of the simple expressions is found to be false, any further simple expressions are not evaluated. Likewise, if the compound expression is an OR expression and one of the simple expressions is found to be true, any further simple expressions are not evaluated. In these cases, the compound expression is either false or true, respectively, and the condition codes reflect the result of the last simple expression evaluated.

C.4.3.5 Statement Formatting

The format of structured control statements differs somewhat from normal assembler usage. Whereas a standard assembler line is split into fields separated by blanks or tabs with no white space inside the fields, structured control statement formats vary depending on the statement being analyzed. In general, all structured control directives are placed in the opcode field (with an optional label in the label field) and white space separates all distinct fields in the statement. Any structured control statement may be followed by a comment on the same logical line.

C.4.3.6 Expression Formatting

Given an expression of the form

```
          <op1>  <LT>  <op2>  OR  <op3>  <GE>  <op4>
```

there must be white space (blank, tab) between all operands and their associated operators, including boolean operators in compound expressions. Moreover, there must be white space between the structured control directive and the expression, and between the expression and any optional directive modifier (THEN, DO). An assembler expression

used as an operand in a structured control statement expression must not have white space in it, since it is parsed by the standard assembler evaluation routines:

```
.IF     #@CVI(@SQT(4.0))  <GT>  #2; no white space in first operand
```

C.4.3.7 .FOR/.LOOP Formatting

The .FOR and .LOOP directives represent special cases. The .FOR structured control statement consists of several fields:

```
.FOR  <op1>  =  <op2>  TO  <op3>  BY  <op4>  DO
```

There must be white space between all operands and other syntactic entities such as =, TO, BY, and DO. As with expression formatting, an assembler expression used as an operand must not have white space in it:

```
.FOR  X:CNT  =  #0  TO  Y:(targ*2)+1  BY  #@CVI(@POW(2.0,@CVF(R)))
```

In the example above, the .FOR loop operands represented as assembler expressions (symbol, function) do not have embedded white space, whereas the loop operands are always separated from structured control statement keywords by white space.

The count field of a .LOOP statement must be separated from the .LOOP directive by white space. The count itself may be any arbitrary assembler expression and therefore must not contain embedded blanks.

C.4.4 Assembly Listing Format

Structured control statements begin with the directive in the opcode field; any optional label is output in the label field. The rest of the statement is left as is in the operand field, except for any trailing comment; the X and Y data movement fields are ignored. Comments following the statement are output in the comment field (unless the unreported comment delimiter is used).

Statements are expanded using the macro facilities of the assembler. Thus the generated code can be sent to the listing by specifying the MEX assembler option, either via the OPT directive or the -O command line option.

C.4.5 Effects on the Programmer’s Environment

During assembly, global labels beginning with “Z_L” are generated. They are stored in the symbol table and should not be duplicated in user-defined labels. Because these non-local labels ordinarily are not visible to the programmer, there can be problems when local (underscore) labels are interspersed among structured control statements. The SCL option

(see the OPT directive) causes the assembler to maintain the current local label scope when a structured control statement label is encountered.

In the.FOR loop, <op1> is a user-defined symbol. When exiting the loop, the memory/register assigned to this symbol contains the value which caused the exit from the loop.

A compare instruction is produced by the assembler whenever two operands are tested in a structured statement. At runtime, these assembler-generated instructions set the condition codes of the CCR (in the case of a loop, the condition codes are set repeatedly). Any user-written code either within or following a structured statement that references CCR directly (move) or indirectly (conditional jump/transfer) should be attentive to the effect of these instructions.

Jumps or branches generated by structured control statements are forced long because the number and address of intervening instructions between a control statement and its termination are not known by the assembler. The programmer may circumvent this behavior by use of the SCSJMP directive. In all structured control statements except those using only a single condition code expression, registers are used to set up the required counters and comparands. In some cases, these registers are effectively reserved; the .FOR loop uses two data registers to hold the step and target values, respectively, and performs no save/restore operations on these registers. The assembler, in fact, does no save/restore processing in any structured control operation; it simply moves the operands into appropriate registers to execute the compare. The SCSREG directive may be used to reassign structured control statement registers. The MEX assembler option (see the OPT directive) may be used to send the assembler-generated code to the listing file for examination of possible register use conflicts.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

P

Q

R

S

U

W

X

Y